НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

"На правах рукопису"
УДК_____

«До захисту допущено»
Завідувач кафедри
_____ О.В. Коваль
(підпис) (ініціали, прізвище)
"___"_____2018р.

Магістерська дисертація

зі спеціальності 121 Інженерія програмного забезпечення
за спеціалізацією Програмне забезпечення розподілених систем
на тему " Інструментальні засоби прискорення тестування для мобільних додатків "

Виконав: студент 6 курсу, групи _____
Зубарчук Олександра Дмитрівна _____
(прізвище, ім'я, по батькові) (підпис)

Науковий керівник к.т.н, доцент Варава І. А. _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент_____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2018

Національний технічний університет України
“ Київський політехнічний інститут ім. Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти другий, магістерський

зі спеціальності - 121 Інженерія програмного забезпечення
за спеціалізацією - Програмне забезпечення розподілених систем

ЗАТВЕРДЖУЮ
Завідувач кафедри
Коваль О.В. _____
(прізвище, ініціали) (підпис)
«____» _____ 2018р.

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ ДИСЕРТАЦІЮ СТУДЕНТУ

Зубарчук Олександрі Дмитрівні
(прізвище, ім’я, по батькові)

1. Тема дисертації Інструментальні засоби прискорення тестування мобільних додатків

Науковий керівник к.т.н. Варава Іван Андрійович
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “___”_____20___року №___

2. Строк подання студентом дисертації _____

3. Об’єкт дослідження Засоби автоматизованого тестування програмного забезпечення

4. Предмет дослідження Вирішення проблеми швидкості виконання автоматизованого тестування мобільних додатків

5. Перелік питань, які потрібно розробити
1) Аналіз сучасних методів тестування мобільних додатків.
2) Інструментальні засоби створення систем автоматизованого тестування.
3) Методика прискорення автоматизованого тестування мобільних додатків.

6. Орієнтований перелік ілюстративного матеріалу презентація PowerPoint на 17 слайдів

7. Орієнтований перелік публікацій: 1 публікація, враховуючи участь у конференції

_____

8. Дата видачі завдання «____»_____201___р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання магістерської дисертації	Строки виконання етапів магістерської дисертації	Примітка
1	Затвердження теми роботи	17.05.2018	
2	Вивчення та аналіз задачі. Проведення дослідження по вибраній темі	01.06.2018-03.09.2018	
3	Розробка архітектури та загальної структури системи	03.09.2018-28.09.2018	
4	Програмна реалізація системи	01.10.2018-26.10.2018	
5	Захист програмного продукту	22.10.2018	
6	Оформлення пояснювальної записки	02.09.2018-10.12.2018	
7	Передзахист	26.11.2018- 30.11.2018	
8	Захист	17.12.2018	

Студент _____ Зубарчук О. Д. _____
(підпис) (прізвище та ініціали)

Науковий керівник _____ Варава І. А._____
(підпис) (прізвище та ініціали)

РЕФЕРАТ
НА МАГІСТЕРСЬКУ ДИСЕРТАЦІЮ

виконану на тему “Інструментальні засоби прискорення тестування мобільних додатків”

студентом Зубарчук Олександрою Дмитрівною

Структура та обсяг дипломної роботи. Магістерська дисертація складається зі вступу, чотирьох розділів, висновку, переліку посилань з 36 найменувань, 5 додатків, і містить 15 рисунків, 3 таблиць. Повний обсяг магістерської дисертації складає 92 сторінок.

Актуальність теми. Основна задача автоматизації тестування мобільних додатків - скорочення витрат на випробування програми після її модернізації та позбавлення при перевірках людського фактору. Проте, досить часто, швидкість виконання автоматизованих тестів, бажає бути кращою. Більшість великих компаній, що розробляють мобільні додатки, змушені запускати тести з вечора, щоб побачити результат регресійного тестування зранку. Така кількість часу, витрачена на тестування, не завжди є доступною, особливо, якщо необхідно терміново виправити помилку в роботі мобільного додатку. За таких обставин, компанії, що розробляють мобільні додатки на маленькі або середні масштаби, нехтують прогонкою автоматизованих тестів.

Мета дослідження полягає створенні методики, на основі наявних інструментальних та програмних засобів, для прискорення тестування мобільних додатків.

Для досягнення поставленої задачі були сформульовані наступні завдання дослідження, що визначили логіку дослідження та його структуру:

1. Вивчити та проаналізувати літературу з проблем прискорення автоматизованого тестування;
2. Зробити аналіз засобів автоматизованого тестування;

3. Об’єднати та модифікувати доступні інструментальні засоби для прискорення тестування мобільних додатків;

4. На основі обраних засобів, розробити методику для прискорення тестування мобільних додатків;

5. Розробити автоматизовані тести використовуючи створену методику;

6. Зробити аналіз отриманих результатів.

Об’єктом дослідження є засоби автоматизованого тестування програмного забезпечення.

Предметом дослідження є вирішення проблеми швидкості виконання автоматизованого тестування мобільних додатків.

Наукова новизна одержаних результатів. Наукова новизна полягає в удосконаленні методів автоматизованого тестування мобільних додатків, пришвидшивши їх та скоротити витрати на тестування додатку.

Практичне значення одержаних результатів роботи полягає в розробці методики з описом кроків для подальшої розробки системи автоматизованого тестування будь-якого мобільного додатку.

Публікації. Мобільний додаток для підвищення розумової діяльності шляхом фізичних навантажень / О.Д. Зубарчук, В.В. Козяр // Підсумки розвитку наукової думки: 2018, Збірник тез, м. Івано-Франківськ, Україна, 5 грудня.

Ключові слова. Автоматизація тестування, прискорення тестування, мобільний додаток, тестування програмного забезпечення.

РЕФЕРАТ
НА МАГИСТЕРСКУЮ ДИСЕРТАЦИЮ

выполненную на тему "Инструментальные средства ускорения тестирования мобильных приложений"

студенкой Зубарчук Александрой Дмитриевной

Структура и объем дипломной работы. Магистерская диссертация состоит из введения, четырех глав, заключения, списка ссылок из 36 наименований, 5 приложений, и содержит 15 рисунков, 3 таблицы. Полный объем магистерской диссертации составляет 92 страниц.

Актуальность темы. Основная задача автоматизации тестирования мобильных приложений - сокращение расходов на испытания программы после ее модернизации и лишения при проверках человеческого фактора. Однако, достаточно часто, скорость выполнения автоматизированной х тестов, оставляет желать лучшего. Большинство крупных компаний, разрабатывающих мобильные приложения, вынуждены запускать тесты с вечера, чтобы увидеть результат регрессионного тестирования утром. Такое количество времени, затраченное на тестирование, не всегда доступна, особенно, если необходимо срочно исправить ошибку в работе мобильного приложения. При таких обстоятельствах, компании, разрабатывающие мобильные приложения на маленькие или средние масштабы, пренебрегают прогонкой автоматизированных тестов.

Цель исследования заключается создании методики, на основе имеющихся инструментальных и программных средств, для ускорения тестирования мобильных приложений.

Для достижения поставленной задачи были сформулированы следующие задачи исследования, определили логику исследования и его структуру:

1. Изучить и проанализировать литературу по проблемам ускорения автоматизированного тестирования;

2. Сделать анализ средств автоматизированного тестирования;

3. Объединить и модифицировать доступны инструментальные средства для ускорения тестирования мобильных приложений;

4. На основе выбранных средств, разработать методику для ускорения тестирования мобильных приложений;

5. Разработать автоматизированные тесты используя созданную методику;

6. Сделать анализ полученных результатов.

Объектом исследования являются средства автоматизированного тестирования программного обеспечения.

Предметом исследования является решение проблемы скорости выполнения автоматизированного тестирования мобильных приложений.

Научная новизна полученных результатов. Научная новизна а заключается в совершенствовании методов автоматизированного тестирования мобильных приложений, ускорив их и сократить затраты на тестирование приложения.

Практическое значение работы заключается в разработке методики с описанием шагов для дальнейшей разработки системы автоматизированного тестирования любого мобильного приложения.

Публикации. Мобильное приложение для повышения умственной деятельности путем физических нагрузок / А.Д. Зубарчук, В.В. Козяр // Итоги развития научной мысли: 2018, Сборник тезисов., Г. Ивано-Франковск, Украина, 5 декабря.

Ключевые слова. Автоматизация тестирования, ускорение тестирования, мобильное приложение, тестирование программного обеспечения.

ABSTRACT

FOR MASTER DISORDERSHIP

performed on the topic "Tools for accelerating the testing of mobile applications"

student Zubarchuk Olexandra

The structure and scope of the thesis. The master thesis consists of introduction, four chapters, conclusion, list of references of 36 titles, 5 appendices, and contains 15 figures, 3 tables. The full volume of the master's thesis is 92 pages.

Relevance of the topic. The main task of automating the testing of mobile applications is to reduce the cost of testing the program after its modernization and deprivation of human factor checks. However, quite often, the speed of performing automated x tests leaves much to be desired. Most large companies developing mobile applications are forced to run tests in the evening to see the result of regression testing in the morning. This amount of time spent on testing is not always available, especially if it is urgently necessary to correct an error in the operation of a mobile application. Under such circumstances, companies developing mobile applications on a small or medium scale, neglect the sweep of automated tests.

The purpose of the research is to create a methodology, based on the available tools and software, to accelerate the testing of mobile applications.

To achieve the task, the following research tasks were formulated, defined the research logic and its structure:

1. To study and analyze the literature on the acceleration of automated testing;

2. Make an analysis of automated testing tools;

3. Combine and modify available tools to speed up the testing of mobile applications;

4. On the basis of the means chosen, develop a methodology for accelerating the testing of mobile applications;

5. Develop automated tests using the created methodology;

6. Make an analysis of the results.

The object of the study are automated software testing tools.

The subject of the research is to solve the problem of the speed of automated testing of mobile applications.

Scientific novelty of the results. Scientific novelty is to improve the methods of automated testing of mobile applications, speeding them up and reduce the cost of testing the application.

The practical significance of the work lies in the development of a methodology with a description of steps for the further development of an automated testing system for any mobile application.

Publications. Mobile application for enhancing mental activity through physical exertion / A.D. Zubarchuk, V.V. Kozyar // Results of the development of scientific thought: 2018, Abstracts collection., G. Ivano-Frankivsk, Ukraine, December 5.

Keywords. Test automation, test acceleration, mobile application, software testing.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПП – програмний продукт

ПЗ – програмне забезпечення

ОС – операційна система

JVM – Java Virtual Machine

SDK – Software Developer Kit

JIT – Just In Time

API – Application Programming Interface

ERD – Entity–relationship Diagram

SADT – Structured analysis and design technique

DFD – Data Flow Diagram

UML – Unified Modeling Language

XML – Extensible Markup Language

ВСТУП

У процесі взаємодії людина – комп’ютер, шукаючи інформацію або даючи команди з застосовуванням елементів графічного інтерфейсу, можна встановлювати безпосередній візуальний контакт з ними. Вагомим для цього процесу є автоматизація тестування. Скрипти та макроси, які контролюють елементи графічного інтерфейсу, або звертаються до елементу за назвою, яке може бути стороннім або навіть недоступним для користувача, або областю на екрані, що може бути змінюваною.

Дана робота присвячена питанням автоматизованого тестування [1] користувацького інтерфейсу, а саме швидкої розробки автоматизованих тестів для мобільних додатків та методах, за допомогою яких можна прискорити їх виконання.

На основі вищесказаного була сформована мета роботи: створення методології, на основі наявних інструментальних та програмних засобів, для прискорення тестування мобільних додатків.

Задачі, що були поставлені для досягнення мети роботи:

7. Вивчити та проаналізувати літературу з проблем прискорення автоматизованого тестування;
8. Зробити аналіз засобів автоматизованого тестування;
9. Об’єднати та модифікувати доступні інструментальні засоби для прискорення тестування мобільних додатків;
10. На основі обраних засобів, розробити методику для прискорення тестування мобільних додатків;
11. Розробити автоматизовані тести використовуючи створену методику;
12. Зробити аналіз отриманих результатів.

Розроблена методика має дозволяти набагато швидше розробляти автоматизовані тести, використовуючи додатковий рівень абстракції, що дозволить користувачам встановлювати зв’язок об’єктів з елементами графічного інтерфейсу, що також дає можливість прискорити виконання автоматизації.

Розробка методики для автоматизованого тестування мотивована відсутністю ефективного за швидкістю механізму створення тестів графічного інтерфейсу та їх виконання. Можливість швидкої побудови об’єктної моделі для довільних елементів графічного інтерфейсу має вирішальне значення, для програмістів, що створюють автоматизовані тести. Також великою проблемою є швидкість виконання автоматизованих тестів звичними засобами. В моменти релізу нових версій мобільних додатків час, витрачений на тестування версії та перевірки правильності функціонування усіх елементів, відіграє, після якості самого тестування, першочергову роль.

РОЗДІЛ 1 ОГЛЯД І АНАЛІЗ ТЕОРЕТИЧНИХ ВІДОМОСТЕЙ

Тестування програмного забезпечення (англ. Software Testing) — це процес технічного дослідження, призначений для виявлення інформації про якість продукту відносно контексту, в якому він має використовуватись. Техніка тестування також включає як процес пошуку помилок або інших дефектів, так і випробування програмних складових з метою оцінки. Може оцінюватись:

- відповідність вимогам, якими керувалися проектувальники та розробники
- правильна відповідь для усіх можливих вхідних даних
- виконання функцій за прийнятний час
- практичність
- сумісність з програмним забезпеченням та операційними системами
- відповідність задачам замовника.

Оскільки число можливих тестів навіть для нескладних програмних компонент практично нескінченне, тому стратегія тестування полягає в тому, щоб провести всі можливі тести з урахуванням наявного часу та ресурсів. Як результат програмне забезпечення (ПЗ) тестується стандартним виконанням програми з метою виявлення баґів (помилок або інших дефектів).

Тестування ПЗ може надавати об'єктивну, незалежну інформацію про якість ПЗ, ризики відмови, як для користувачів так і для замовників [1].

Тестування може проводитись, як тільки створено виконуваний код (навіть частково завершено). Процес розробки зазвичай передбачає коли та як буде відбуватися тестування. Наприклад, при поетапному процесі, більшість тестів відбувається після визначення системних вимог і тоді вони реалізуються в тестових програмах. На противагу цьому, відповідно до вимог гнучкої розробки ПЗ, програмування і тестування часто відбувається одночасно.

Тестування програмного забезпечення — техніка контролю якості, що перевіряє відповідність між реальною і очікуваною поведінкою програми завдяки кінцевому набору тестів, які обираються певним чином.

Якість не є абсолютною, це суб'єктивне поняття. Тому тестування, як процес своєчасного виявлення помилок та дефектів, не може повністю забезпечити коректність програмного забезпечення [21]. Воно тільки порівнює стан і поведінку продукту зі специфікацією. При цьому треба розрізняти тестування програмного забезпечення й забезпечення якості програмного забезпечення, до якого належать всі складові ділового процесу, а не тільки тестування.

Зазвичай, поняття якості обмежується такими поняттями як коректність, надійність, практичність, безпечність, але може містити більше технічних вимог, котрі описані у стандарті ISO 9126. Склад та зміст супутньої документації процесу тестування визначається стандартом IEEE 829—1998 Standard for Software Test Documentation. Існує багато підходів до тестування програмного забезпечення, але ефективне тестування складних продуктів — це по суті дослідницький та творчий процес, а не тільки створення та виконання рутинної процедури.

Перші програмні системи розроблялися в рамках програм наукових досліджень або програм для потреб міністерств оборони. Тестування таких продуктів проводилося строго формалізовано із записом всіх тестових процедур, тестових даних, отриманих результатів [29]. Тестування виділялося в окремий процес, який починався після завершення кодування, але при цьому, як правило, виконувалося тим же персоналом.

У 1960-х багато уваги приділялося «вичерпному» тестуванню, яке повинно проводитися з використанням усіх шляхів у коді або всіх можливих вхідних даних. Було відзначено, що в цих умовах повне тестування ПЗ неможливе, тому що, по-перше, кількість можливих вхідних даних дуже велика, по-друге, існує безліч шляхів, по-третє, складно знайти проблеми в архітектурі та специфікаціях. З цих причин «вичерпне» тестування було відхилено й визнано теоретично неможливим.

На початку 1970-х тестування ПЗ розглядалося як «процес, спрямований на демонстрацію коректності продукту» або як «діяльність з підтвердження правильності роботи ПЗ». У програмній інженерії, яка в той час зароджувалася, верифікація ПЗ визначалася як «доказ правильності». Хоча концепція була теоретично перспективною, на практиці вона вимагає багато часу й не охоплювала всі аспекти тестування. Було вирішено, що доказ правильності — неефективний метод тестування ПЗ [24]. Однак, у деяких випадках демонстрація правильної роботи використовується і в наші дні, наприклад, приймально-здавальні випробування. У другій половині 1970-х тестування представляється як виконання програми з наміром знайти помилки, а не довести, що вона працює. Успішний тест — це тест, який виявляє раніше невідомі проблеми. Даний підхід цілком протилежний попередньому. Зазначені два визначення являють собою «парадокс тестування», в основі якого лежать два протилежних твердження: з одного боку, тестування дозволяє переконатися, що продукт працює добре, а з іншого — виявляє помилки у ПЗ, показуючи, що продукт не працює. Друга мета тестування є більш продуктивною з точки зору поліпшення якості, оскільки не дозволяє ігнорувати недоліки ПЗ.

У 1980-х тестування розширилося таким поняттям як запобіганням дефектам. Проектування тестів — найбільш ефективний з відомих методів запобігання помилок. В цей же час почали висловлюватися думки, що необхідна методологія тестування, зокрема, що тестування повинно включати перевірки впродовж усього циклу розроблення, при цьому це має бути керований процес. В ході тестування треба перевірити не тільки зібрану програму, але й вимоги, код, архітектуру, самі тести. «Традиційне» тестування, яке існувало до початку 1980-х, відносилося тільки до скомпілювати, готової системі (зараз це зазвичай називається системне тестування), але надалі тестувальники стали залучатися в усі аспекти життєвого циклу розроблення [31]. Це дозволяло раніше знаходити проблеми у вимогах та архітектурі й тим самим скорочувати терміни та бюджет розроблення. У середині 1980-х з'явилися перші інструменти для автоматизованого тестування. Передбачалося, що комп'ютер зможе виконати більше тестів, ніж людина, причому

зробить це більш надійно. Спочатку ці інструменти були вкрай простими й не мали можливості написання сценаріїв на скриптових мовах.

На початку 1990-х у поняття «тестування» стали включати планування, проектування, створення, підтримку й виконання тестів та тестових оточень, а це означало перехід від тестування до забезпечення якості, що охоплює весь цикл розроблення ПЗ. У цей час починають з'являтися різні програмні інструменти для підтримки процесу тестування: більш просунуті середовища для автоматизації з можливістю створення скриптів і генерації звітів, системи управління тестами, ПЗ для проведення навантажувального тестування. У середині 1990-х з розвитком Інтернету й розробленням великої кількості веб-застосунків особливої популярності стало набувати «гнучке тестування» (за аналогією з гнучкими методологіями програмування).

У 2000-х з'явилося ще більш широке визначення тестування, коли в нього було додано поняття «оптимізація бізнес-технологій». ВТО направляє розвиток інформаційних технологій згідно з цілями бізнесу. Основний підхід полягає в оцінці та максимізації значущості всіх етапів життєвого циклу розроблення ПЗ для досягнення необхідного рівня якості, продуктивності, доступності.

1.1 Основні поняття та визначення тестування програмного забезпечення

Тестування — це одна з технік контролю якості, що включає в себе:

- Планування робіт (Test Management)
- Проектування тестів (Test Design)
- Виконання тестування (Test Execution)
- Аналіз отриманих результатів (Test Analysis).

Верифікація (Verification) — це процес оцінки системи або її компонентів з метою визначити чи задовольняють результати поточного етапу розробки умовам, сформованим на початку цього етапу. Тобто чи виконуються цілі, терміни, завдання з розробки проекту, визначені на початку поточної фази. Валідація (Validation) — це визначення відповідності розроблюваного програмного забезпечення між очікуваннями і потребами користувача, вимогам до системи.

План Тестування (Test Plan) — це документ, що описує весь обсяг робіт з тестування, починаючи з опису об'єкта, стратегії, розкладу, критеріїв початку і закінчення тестування, до необхідного в процесі роботи обладнання, спеціальних знань, а також оцінки ризиків з варіантами їх вирішення.

Тест дизайн (Test Design) — це етап процесу тестування програмного забезпечення, на якому проектуються і створюються тестові випадки (тест кейси), відповідно до визначених раніше критеріями якості та цілями тестування.

Тестовий випадок (Тест кейс/Test Case) — це документ, що описує сукупність кроків, конкретних умов і параметрів, необхідних для перевірки реалізації тестованої функції або її частини.

Баг/Дефект Репорт (Bug Report) — це документ, що описує ситуацію або послідовність дій (Steps), що призвела до некоректної роботи об'єкта тестування (Misbehavior), із зазначенням причин та очікуваного результату (Expected Result).

Тестове Покриття (Test Coverage) — це одна з метрик оцінки якості тестування, що представляє із себе щільність покриття тестами вимог або коду, що виконується.

Деталізація Тест Кейсів (Test Case Specification) — це рівень деталізації опису тестових кроків і необхідного результату, при якому забезпечується розумне співвідношення часу проходження до тестового покриття.

Час Проходження Тест Кейса (Test Case Pass Time) — це час від початку проходження кроків тест кейса до отримання результату тесту.

1.2 Методи тестування програмного забезпечення

1.2.1 Статичне та динамічне тестування

Тестова діяльність, що пов'язана з аналізом результатів розробки програмного забезпечення, називається статичним тестуванням. Воно передбачає перевірку програмних кодів, контроль та перевірку програми без запуску на комп'ютері [32]. Тестова діяльність, що передбачає експлуатацію програмного продукту, називається динамічним тестуванням. Динамічне та статичне тестування доповнюють одне одного.

На етапі статичного тестування перевіряється вся документація, отримана як результат життєвого циклу програми. Це і технічне завдання, і специфікація, і вихідний текст програми на мові програмування. Вся документація аналізується на предмет дотримання стандартів програмування. У результаті статичної перевірки встановлюється, наскільки програма відповідає заданим критеріям та вимогам замовника. Усунення неточностей та помилок у документації — запорука того, що створюваний програмний засіб має високу якість.

Динамічні методи застосовуються в процесі безпосереднього виконання програми. Коректність програмного засобу перевіряється на безлічі тестів або наборів підготовлених вхідних даних. При прогоні кожного тесту збираються та аналізуються дані про відмови та збої в роботі програми.

1.2.2 Тестування «білої скриньки»

При тестуванні за принципом «білої скриньки» тестувальнику відома внутрішня структура програми, досліджуються внутрішні елементи програми і зв'язки між ними.

Об'єктом тестування тут є не зовнішня, а внутрішня поведінка програми. Перевіряється коректність побудови всіх елементів програми та правильність їхньої взаємодії один з одним. Зазвичай аналізуються керуючі зв'язки елементів, рідше — інформаційні зв'язки. Тестування за принципом «білого ящика» характеризується ступенем, в якому тести виконують або покривають логіку (вихідний текст) програми [32].

Зазвичай тестування «білої скриньки» засноване на аналізі керуючої структури програми. Програма вважається повністю перевіреною, якщо проведено вичерпне тестування маршрутів (шляхів) її графа управління.

У цьому випадку формуються тестові варіанти, в яких:

- Гарантується перевірка всіх незалежних маршрутів програми.
- Знаходяться гілки True, False для всіх логічних рішень.
- Виконуються всі цикли (у межах їхніх кордонів та діапазонів).
- Аналізується правильність внутрішніх структур даних.

Недоліки тестування «білої скриньки»:

- Кількість незалежних маршрутів може бути дуже велика.
- Повне тестування маршрутів не гарантує відповідності програми вихідним вимогам до неї.
- У програмі можуть бути пропущені деякі маршрути.
- Не можна виявити помилки, поява яких залежить від даних.

Переваги тестування «білої скриньки» пов'язані з тим, що принцип «білої скриньки» дозволяє врахувати особливості програмних помилок:

- Кількість помилок мінімально в «центрі» і максимально на «периферії» програми.
- Попередні припущення про ймовірність потоку керування або даних у програмі часто бувають некоректними. У результаті типовим може стати маршрут, модель обчислень за яким опрацьована слабо.

- При записі алгоритму програмного забезпечення у вигляді тексту на мові програмування можливе внесення типових помилок трансляції (синтаксичних та семантичних).
- Деякі результати в програмі залежать не від вихідних даних, а від внутрішніх станів програми.

Кожна з цих причин є аргументом для проведення тестування за принципом «білої скриньки». Тести «чорної скриньки» не можуть реагувати на помилки таких типів.

1.2.3 Тестування «чорної скриньки»

При тестуванні за принципом «чорної скриньки» тестувальнику відомі функції програми, досліджується робота кожної функції на всій області визначення.

Основне місце програми тестів «чорної скриньки» — інтерфейс ПЗ.

Ці тести демонструють:

- Як виконуються функції програми.
- Як приймаються вихідні дані.
- Як виробляються результати.
- Як зберігається цілісність зовнішньої інформації.

При тестуванні «чорної скриньки» розглядаються системні характеристики програм, ігнорується їхня внутрішня логічна структура. Вичерпне тестування, як правило, неможливе. Наприклад, якщо в програмі 10 вхідних величин і кожна приймає по 10 значень, то кількість тестових варіантів становитиме 10^10 . Тестування «чорної скриньки» не реагує на багато особливостей програмних помилок.

Тестування «чорної скриньки» (функціональне тестування) дозволяє отримати комбінації вхідних даних, які забезпечують повну перевірку всіх функціональних вимог до програми [32]. Програмний виріб тут розглядається як «чорна скринька», чию поведінку можна визначити тільки дослідженням його входів та відповідних виходів. При такому підході бажано мати:

- Набір, утворений такими вхідними даними, які призводять до аномалій у поведінці програми (назвемо його ІТ);
- Набір, утворений такими вхідними даними, які демонструють дефекти програми (назвемо його ОТ).

Будь-який спосіб тестування «чорної скриньки» повинен:

- Виявити такі вхідні дані, які з високою ймовірністю належать набору ІТ;
- Сформулювати такі очікувані результати, які з високою імовірністю є елементами набору ОТ.

Принцип «чорної скриньки» не альтернативний принципу «білої скриньки». Скоріше це доповнює підхід, який виявляє інший клас помилок.

Тестування «чорної скриньки» забезпечує пошук наступних категорій помилок:

- Некоректних чи відсутніх функцій;
- Помилок інтерфейсу;
- Помилок у зовнішніх структурах даних або в доступі до зовнішньої бази даних;
- Помилок характеристик (необхідна ємність пам'яті і т. д.);
- Помилок ініціалізації та завершення.

Подібні категорії помилок способами «білої скриньки» не виявляються.

1.3 Види тестування програмного забезпечення

Усі види тестування програмного забезпечення можна розділити за наступними ознаками [34].

За ступенем автоматизації:

- Ручне тестування (manual testing);
- Автоматизоване тестування (automated testing);
- Напівавтоматизоване тестування (semiautomated testing).

За ступенем підготовленості до тестування:

- Тестування по документації (formal testing);
- Тестування ad hoc або інтуїтивне тестування (ad hoc testing) — тестування без тест плану та документації, що базується на методиці передбачення помилки на власному досвіді тестувальника.

За знанням системи:

- Тестування чорної скриньки (black box);
- Тестування білої скриньки (white box);
- Тестування сірої скриньки (grey box).

За ступенем ізольованості компонентів:

- Компонентне (модульне) тестування (component/unit testing);
- Інтеграційне тестування (integration testing);
- Системне тестування (system/end-to-end testing).

За часом проведення тестування:

- Альфа-тестування (alpha testing);
- Тестування при прийманні або Димове тестування (smoke testing);
- Тестування нової функціональності (new feature testing);

- Регресивне тестування (regression testing);
- Тестування при здачі (acceptance testing);
- Бета-тестування (beta testing).

За об'єктом тестування:

- Функціональне тестування (functional testing);
- Тестування продуктивності (performance testing);
- Навантажувальне тестування (load testing);
- Стрес-тестування (stress testing);
- Тестування стабільності (stability/endurance/soak testing);
- Тестування зручності використання або Юзабіліті-тестування (usability testing);
- Тестування інтерфейсу користувача (UI testing);
- Тестування безпеки (security testing);
- Тестування локалізації (localization testing);
- Тестування сумісності (compatibility testing).

За ознакою позитивності сценаріїв:

- Позитивне тестування (positive testing);
- Негативне тестування (negative testing).

1.4 Опис видів тестування програмного забезпечення

Інсталяційне тестування

Інсталяційне тестування запевняє, що система встановлена правильно і коректно працює на апаратному забезпеченні конкретного клієнта.

Інсталяційне тестування направлене на перевірку вдалої інсталяції та налаштування, а також оновлення або видалення ПП. На даний момент найбільш поширене тестування встановлення ПП за допомогою інсталяторів.

Тестування сумісності

Основною метою якого є перевірка коректної роботи продукту в певному середовищі. Середовище може включати в себе наступні елементи:

- Аппаратна платформа
- Мережеві пристрої
- Периферія (принтери, CD/DVD-приводи, веб-камери та ін.);
- Операційна система (Unix, Windows, MacOS, …)
- Бази даних (Oracle, MS SQL, MySQL, …)
- Системне програмне забезпечення (веб-сервер, фаєрвол, антивірус, …)
- Браузери (Internet Explorer, Firefox, Opera, Chrome, Safari)

Смоук тестування (Smoke testing)

Мінімальний набір тестів на явні помилки. Цей тест зазвичай виконується самим програмістом. Програму, що не пройшла такий тест, не має сенсу передавати на глибше тестування. Наприклад, якщо програма не пройшла інсталяцію або не може успішно під ’єднатися до бази даних.

Регресивне тестування

Виявляє помилки у вже протестованих ділянках початкового коду. Такі помилки — коли після внесення змін до програми перестає працювати те, що мало б працювати, — називають регресивними помилками.

Регресивне тестування (за деякими джерелами) включає:

- new bug-fix — перевірка виправлення знайдених дефектів;

- old bug-fix — перевірка, що виявлені раніше й виправлені дефекти не відтворюються в системі знову;
- side-effect — перевірка того, що не порушилася працездатність працюючої раніше функціональності, якщо її код міг бути зачеплений під час виправлення деяких дефектів в іншій функціональності.

Функціональне тестування

Перевіряє, чи реалізовані функціональні вимоги, тобто можливості ПЗ в певних умовах вирішувати завдання, потрібні користувачам. Функціональні вимоги визначають, що саме робить продукт, які завдання вирішує.

Функціональні вимоги включають в себе:

- Функціональна придатність
- Точність
- Можливість до взаємодії
- Відповідність стандартам та правилам
- Захищеність

Нефункціональне тестування

Описує тести, необхідні для визначення характеристик ПЗ, які можуть бути виміряні різними величинами. В цілому, це тестування того, «як» система працює. Далі перелічені основні види не функціональних тестів:

- Всі види тестування продуктивності:
- навантажувальне тестування;
- стрессове тестування;
- тестування стабільності та надійності;
- об'ємне тестування;

- Інсталяційне тестування;
- Тестування зручності користування;
- Тестування на «відмову» та відновлення;
- Конфігураційне тестування.

Деструктивне тестування

Намагається привести ПЗ чи підсистему до збою. Воно перевіряє, чи ПЗ продовжує функціонувати навіть при отриманні неправильних або неочікуваних вхідних даних, встановлюючи тим самим надійність перевірки вхідних даних і управління помилками підпрограм.

Тестування швидкодії

Проводиться з метою встановлення, як швидко працює система або її частина, під певним навантаженням. Також може слугувати для перевірки й підтвердження інших атрибутів якості системи, таких як масштабування, надійність та використання ресурсів.

В тестуванні швидкодії виділяють такі напрямки:

- навантажувальне;
- стресс;
- тестування стабільності;
- конфігураційне.

Тестування зручності використання

Виконується з метою визначення зручності використання ПЗ для його подальшого застосування. Це метод оцінки зручності продукту у використанні, заснований на залученні користувачів як тестувальників, випробувачів і підсумовуванні отриманих від них висновків.

1.5 Рівні тестування

1.5.1 Модульне тестування

Відноситься до тестів, які перевіряють функціональність певного розділу коду, зазвичай на функціональному рівні. В об'єктно-орієнтованому середовищі, це, як правило, тестування на рівні класу, а мінімальні модульні тести містять у собі конструктори та деструктори.

Такі типи тестів зазвичай пишуться розробниками під час роботи над кодом (стиль «білої скриньки»), щоб впевнитись, що дана функція працює так, як очікувалося. Одна функція може мати кілька тестів, щоб переглянути всі випадки використання коду. Модульне тестування саме по собі не може перевірити функціонування частини ПЗ, а використовується щоб гарантувати, що основні блоки ПЗ працюють незалежно один від одного.

Модульне тестування — це процес розробки ПЗ, що включає в себе синхронізовані застосування широкого спектру для запобігання дефектів та для виявлення стратегій з метою зниження ризиків розробки ПЗ, часу та витрат [19]. Воно виконується розробником ПЗ або інженером, під час будівельної фази життєвого циклу розробки ПЗ. Модульне тестування спрямоване на усунення помилок проектування. Ця стратегія спрямована на підвищення якості одержуваного ПЗ, до такого рівня, як вимагає процес контролю якості.

Залежно від очікуваної організації розробки ПЗ, модульне тестування може включати статичний аналіз коду, аналіз потоку даних аналізу метрик, експертні оцінки коду, аналізу покриття коду та інші методи перевірки ПЗ.

1.5.2 Інтеграційне тестування

Інтеграційне тестування є типом тестування ПЗ, яке прагне перевірити інтерфейси між компонентами від програмного дизайну. Програмні компоненти можуть бути інтегровані як в рамках ітеративного підходу, так і всі разом [23].

Інтеграційне тестування працює над виявленням дефектів у інтерфейсах та взаємодії інтегрованих компонентів (модулів). Воно проводиться до тих пір, поки великі групи тестованих компонентів ПЗ, які відповідають потрібній архітектурі, починають працювати як система.

Рівні інтеграційного тестування:

- компонентний інтеграційний рівень (Component Integration testing). Перевіряється взаємодія між компонентами системи після проведення компонентного тестування;
- системний інтеграційний рівень (System Integration Testing). Перевіряється взаємодія між різними системами після проведення системного тестування.

Підходи до інтеграційного тестування:

- знизу вгору (Bottom Up Integration). Усі низькорівневі модулі, процедури або функції збираються воєдино і потім тестуються. Після чого збирається наступний рівень модулів для проведення інтеграційного тестування. Даний підхід вважається корисним, якщо всі або практично всі модулі розроблюваного рівня готові [27]. Також даний підхід допомагає визначити за результатами тестування рівень готовності додатків;
- зверху вниз (Top Down Integration). У першу чергу тестуються компоненти верхнього рівня ієрархії об'єктів з використанням заглушок замість компонентів більш низького рівня;
- великий вибух («Big Bang» Integration). Усі або практично усі розроблені модулі збираються разом у вигляді закінченої системи або її основної частини й потім проводиться інтеграційне тестування. Такий підхід дуже хороший для збереження часу. Проте, якщо тест кейси та їхні результати записані

неправильно, то сам процес інтеграції дуже ускладниться, що стане перепоною для команди тестування при досягненні основної мети інтеграційного тестування.

1.5.3 Системне тестування

Тестує інтегровану систему для перевірки відповідності всім вимогам. Крім того, системне тестування ПЗ повинно гарантувати, що програма працює так, як очікувалося, а також, що її не можна знищити або пошкодити її робоче середовище, яке викличе процеси в цьому середовищі, що переведуть систему в неробочий стан. Системне інтеграційне тестування перевіряє, чи система інтегрується в будь-яку зовнішню систему (або системи) відповідно до системних вимог.

- Альфа-тестування — імітація реальної роботи з системою штатними розробниками або реальна робота з системою потенційними користувачами/замовником. Найчастіше альфа-тестування проводиться на ранній стадії розробки продукту, але у деяких випадках може застосовуватися для закінченого продукту як внутрішнього приймального тестування [16]. Іноді альфа-тестування виконується під відлагоджувачем або з використанням середовища, яке допомагає швидко виявляти знайдені помилки. Виявлені помилки можуть бути передані тестувальникам для додаткового дослідження у середовищі, подібному тому, в якому буде використовуватися програма.

- Бета-тестування — у деяких випадках виконується поширення версії з обмеженнями (за функціональністю або часом роботи) для певної групи осіб, з тим щоб переконатися, що продукт містить достатньо мало помилок [3]. Іноді бета-тестування виконується для того, щоб отримати зворотній зв'язок про продукт від його майбутніх користувачів.

Часто для вільного/відкритого ПЗ стадія альфа-тестування характеризує функціональне наповнення коду, а бета-тестування — стадію виправлення

помилок. При цьому, як правило, на кожному етапі розробки проміжні результати роботи доступні кінцевим користувачам.

1.5.4 Тестові скрипти

Тестувальники використовують тестові скрипти на різних рівнях: як у модульному, так і в інтеграційному та системному тестуванні. Тестові скрипти, як правило, пишуться для перевірки компонентів, у яких найбільш висока ймовірність появи відмов або вчасно не знайдена помилка може бути дорогою.

1.5.5 Покриття коду

Покриття коду, за своєю суттю, є тестуванням методом білого ящика. Тестоване ПЗ збирається зі спеціальними налаштуваннями або бібліотеками й/або запускається в особливому середовищі, в результаті чого для кожної використовуваної (виконуваної) функції програми визначається місцезнаходження цієї функції у вихідному коді [35]. Цей процес дозволяє розробникам та фахівцям із забезпечення якості визначити частини системи, які, при нормальній роботі, використовуються дуже рідко або ніколи не використовуються (такі як код обробки помилок тощо). Це дозволяє зорієнтувати тестувальників на тестування найбільш важливих режимів.

Як правило, інструменти та бібліотеки, які використовуються для отримання покриття коду, вимагають значних витрат продуктивності та/або пам'яті, неприпустимих при нормальному функціонуванні ПЗ. Тому вони можуть використовуватися тільки в лабораторних умовах.

1.5.6 Приймальне тестування

Формальний процес тестування, який перевіряє відповідність системи вимогам і проводиться з метою: визначення чи задовольняє система приймальним критеріям; винесення рішення замовником або іншою уповноваженою особою приймається додаток чи ні.

Приймальне тестування виконується на основі набору типових тестових випадків та сценаріїв, розроблених на основі вимог до даного додатку [34]. Рішення про проведення приймального тестування приймається тоді, коли: продукт досяг необхідного рівня якості; замовник ознайомлений з Планом приймальних Робіт (Product Acceptance Plan) або іншим документом, де описаний набір дій, пов'язаних з проведенням приймального тестування, дата проведення, відповідальні тощо.

Фаза приймального тестування триває до тих пір, доки замовник не виносить рішення про відправлення програми на доопрацювання або видачі додатка.

1.6 Автоматизоване тестування програмного забезпечення

Автоматизоване тестування програмного забезпечення — частина процесу тестування на етапі контролю якості в процесі розробки програмного забезпечення. Воно використовує програмні засоби для виконання тестів і перевірки результатів виконання, що допомагає скоротити час тестування і спростити його процес.

Перші спроби «автоматизації» з'явилися в епоху операційних систем DOS і CP/M. Тоді вона полягала у видачі додатком команд через командний рядок і аналізі результатів. Трохи пізніше додалися віддалені виклики через API для роботи з мережі [11]. Вперше про автоматизоване тестування згадується в книзі Фредеріка Брукса «Міфічний людино-місяць», де йдеться про перспективи використання модульного тестування. Але по-справжньому автоматизація тестування стала розвиватися тільки в 1980-х роках.

Існує два основних підходи до автоматизації тестування: тестування на рівні коду і GUI-тестування. До першого типу належить, зокрема, модульне тестування. До другого — імітація дій користувача за допомогою спеціальних тестових фреймворків.

Найпоширенішою формою автоматизації є тестування додатків через графічний інтерфейс користувача [14]. Популярність такого виду тестування пояснюється двома факторами: по-перше, додаток тестується тим же способом, яким його буде використовувати людина, по-друге, можна тестувати додаток, не маючи при цьому доступу до вихідного коду.

1.6.1 GUI-автоматизація

GUI-автоматизація розвивалася протягом 4 поколінь інструментів і технік:

- Утиліти запису і відтворення (capture/playback tools) — записують дії тестувальника під час ручного тестування. Вони дозволяють виконувати тести без прямої участі людини протягом тривалого часу, значно збільшуючи продуктивність і усуваючи «безглузде» повторення одноманітних дій під час ручного тестування [32]. У той же час, будь-яка мала зміна ПЗ, що тестується вимагає перезапису ручних тестів. Тому це перше покоління інструментів не ефективне і не масштабоване.
- Сценарії (Scripting) — форма програмування на мовах, спеціально розроблених для автоматизації тестування ПЗ — пом'якшує багато проблем capture/playback tools. Але розробкою займаються програмісти високого рівня, які працюють окремо від тестувальників, що безпосередньо запускають тести. До того ж скрипти найбільше підходять для тестування GUI і не можуть бути впровадженими, пакетними або взагалі будь-яким чином об'єднані в систему. Нарешті, зміни в ПЗ, яке тестується вимагають складних змін у відповідних

скриптах, і підтримка все більше зростаючої бібліотеки тестуючих скриптів стає зрештою непереборним завданням.

- Data-driven testing — методологія, яка використовується в автоматизації тестування. Особливістю є те, що тестові скрипти виконуються і верифікуються на основі даних, які зберігаються в центральному сховищі даних або БД. Роль БД можуть виконувати ODBC-ресурси, csv або xls файли і т. д. Data-driven testing — це об'єднання декількох взаємодіючих тестових скриптів та їх джерел даних в фреймворк, який використовується в методології. У цьому фреймворку змінні використовуються як для вхідних значень, так і для вихідних перевірочних значень: у тестовому скрипті зазвичай закодовані навігація по додатком, читання джерел даних, ведення логів тестування. Таким чином, логіка, яка буде виконана в скрипті, також залежить від даних.
- Keyword-based автоматизація передбачає поділ процесу створення кейсів на 2 етапи: етап планування та етап реалізації.

1.6.2 Модульне тестування

Модульне тестування (англ. Unit testing) — це метод тестування програмного забезпечення, який полягає в окремому тестуванні кожного модуля коду програми. Модулем називають найменшу частину програми, яку може бути протестованою. У процедурному програмуванні модулем вважають окрему функцію або процедуру. В об'єктно-орієнтованому програмуванні — інтерфейс, клас. Модульні тести, або unit-тести, розробляються в процесі розробки програмістами та, іноді, тестувальниками білої скриньки (white-box testers) [16].

В даній роботі увагу буде зосереджено саме на GUI тестуванні. Для автоматизації тестування існує велика кількість додатків. Деякі з них:

- HP LoadRunner, HP QuickTest Professional, HP Quality Center
- Segue SilkPerformer

- IBM Rational FunctionalTester, IBM Rational PerformanceTester, IBM Rational TestStudio
- SmartBear Software TestComplete

Використання цих інструментів допомагає тестувальникам автоматизувати наступні задачі:

- установка продукту
- створення тестових даних
- GUI-взаємодія
- визначення проблеми

Однак автоматичні тести не можуть повністю замінити ручне тестування. Автоматизація всіх випробувань — дуже дорогий процес, і тому автоматичне тестування є лише доповненням ручного тестування. Найкращий варіант використання автоматичних тестів — регресійне тестування. Інструментальні засоби для прискорення регресійного тестування мобільних додатків будуть описані в подальших розділах роботи.

Висновки до 1 розділу

Вивчено проблеми тестування мобільних додатків. Однією з головних проблем автоматизованого тестування є його трудомісткість: попри те, що воно дозволяє усунути частину рутинних операцій і прискорити виконання тестів, великі ресурси можуть витрачатися на оновлення самих тестів. З іншого боку, при зміні інтерфейсу програми необхідно заново переписати всі тести, які пов'язані з оновленими вікнами, що при великій кількості тестів може відняти значні ресурси.

Другим недоліком є повільне виконання автоматизованих тестів при великому обсязі роботи та великій кількості тестів.

Якщо з першим недоліком боротися можливо лише збільшенням кількості програмістів, для прискорення оновлення тестового покриття, то проблему швидкості виконання вже готових тестів можна вирішити декількома способами. Детальніше дані способи описані в наступних розділах магістерської дисертації.

РОЗДІЛ 2 ІНСТРУМЕНТИ ТЕСТУВАННЯ МОБІЛЬНИХ ДОДАТКІВ

2.1 Особливості тестування мобільних додатків

Тестування додатків на мобільних пристроях в цілому відповідає загальним принципам тестування, але також присутній ряд особливостей, які характерні саме для тестування мобільних додатків.

Для того, щоб зрозуміти особливості тестування прикладних програм на мобільних пристроях, необхідно враховувати фактори, які відрізняють мобільну

програму від десктопної, а саме: специфічні та різноманітні операційні системи для мобільних платформ, різноманітні конфігурації комплектуючих, функціональність таких пристроїв як комунікатори і т. п.

У зв'язку з цими факторами підхід до тестування мобільних додатків досить сильно відрізняється від десктопного [32]. З’являється велика кількість додаткових нюансів і вимог, які необхідно протестувати.

Основні моменти, на які необхідно звертати увагу при тестуванні мобільних додатків наведено нижче.

Розмір екрану та сенсорний інтерфейс:

- Розмір всіх елементів графічного інтерфейсу користувача;
- Перевірка можливостей використання всіх активних елементів (кнопки, посилання і т. п.);
- Швидкість реакції активних елементів повинна бути достатньо високою;
- Перевірка того, що багатократне швидке натискання кнопки не викликає екстреного завершення програми;
- Підтримка мультитачу – одночасне натискання кількох кнопок;
- Підтримка горизонтального (ландшафтного) та вертикального (портретного) положення екрану;
- Перевірка використання в додатку спеціальних жестів.

Ресурси телефону:

- Необхідно проконтролювати можливі втрати пам'яті. Часто це трапляється в програмах із вікнами, що містять велику кількість інформації. Також втрата пам'яті може бути присутня під час тривалої роботи додатку.
- Перевірка обробки ситуацій недостатньої пам'яті для функціонування операційної системи, під час роботи програми в активному та фоновому режимах.
- Недостаток місця для установки або роботи додатка.

- Перевірка роботи батареї (аккумулятора) пристрою при запущеному додатку, роботі в фоновому режимі, при включеному Wi-Fi, 3G та LTE Інтернеті, без підключення до мережі тощо.

Різні розширення екрану та версії ОС:

- Необхідно перевірити роботу програми на пристроях з різними дозволами екрана [1]. На екранах з високим дозволом (наприклад, ретина-экран) елементи інтерфейсу та текст відображаються меншими, при роботі додатків на пристрої з екраном більш низького дозволу, елементи інтерфейсу можуть стати занадто великими.

- Необхідно переконатися, що додаток не може бути встановлений на непідтримувані пристрої. При цьому обов'язково тестування додатків на всіх заявлених підтримуваних пристроях.

Користувач мобільного пристрою очікує, що встановлені ними програми прості, інтуїтивно зрозумілі, працюють завжди і без переривань [3]. Якщо очікування не виправдовуються, то користувач просто встановлює аналогічне додаток від іншого розробника, якого в сфері мобільного розробки завжди достатньо. Тому якість програми є одним з головних факторів його популярності.

2.2 Класифікація інструментів тестування мобільних додатків

Існують різні інструменти тестування мобільних додатків. Умовно їх можна поділити на інструменти для ручного та автоматизованого тестування. Також можна класифікувати за такою схемою:

2.2.1 Ручне і автоматизоване мобільне тестування

Сьогодні багато фахівців підтримують думку про те, що ручне тестування в кінцевому підсумку перестане використовуватися [25]. Обійтися без автоматизації тестування на сьогоднішній день дуже важко, однак є ситуації, коли надають перевагу ручному тестуванню.

Переваги ручного тестування мобільних додатків:

- Це більш економічно вигідно в короткостроковій перспективі.
- Ручне тестування більш гнучке.
- Найкраще моделювання дій користувача.

Недоліки ручного тестування мобільних додатків:

- Ручні тестові приклади важко використовувати повторно.
- Менш ефективне виконання певного постійного завдання.
- Процес тестування повільний.
- Деякі типи тестових випадків не можуть бути виконані вручну (тестування навантаження).

Переваги автоматизованого тестування додатків:

- Процес тестування займає мало часу.
- Економічність в довгостроковій перспективі використання.
- Автоматизовані тестові випадки легко використовувати повторно.
- Єдине рішення для деяких видів тестування (тестування продуктивності).
- Результати випробувань легкодоступні.

Недоліки автоматизованого тестування додатків:

- У деяких мобільних засобів тестування є обмеження.
- Процес тестування займає багато часу.
- Автоматизоване тестування найменш ефективне у визначенні зручності користування.

2.2.2 Симулятори та емулятори

Часто плутають значення слів «емулятор» і «симулятор». Фактично, емулятор – це оригінальна заміна пристрою. У вас немає можливості модифікувати програми і додатки, але ви можете їх запускати [26]. Симулятор, в свою чергу, не копіює апаратне забезпечення пристрою, однак у вас є можливість налаштувати аналогічне середовище, таке як в ОС оригінального пристрою. Таким чином, краще використовувати мобільні симулятори для тестування мобільного додатка. Емулятори більше підходять для тестування мобільних сайтів.

Переваги використання симуляторів для тестування мобільного додатка:

- Просте налаштування.
- Швидка дія.
- Допомагає перевіряти і тестувати поведінку вашого мобільного додатка.
- Економічно вигідно.
- Недоліки використання симуляторів для тестування мобільного додатка:
- Апаратне обладнання не враховується.
- Можливі помилкові спрацьовування.
- Отримання неповних даних про результати моделювання, що створює певні труднощі для повного аналізу результатів тестувань.

2.2.3 Хмарні та онлайн сервіси

Тестування мобільних додатків з використанням хмарних інструментів, мабуть, є оптимальним вибором. Це може допомогти вам уникнути недоліків реальних пристроїв і симуляторів.

Основні переваги цього підходу:

- Легка доступність.
- Можливість запуску мобільних пристроїв на декількох системах.
- Можливість не тільки тестувати, а й оновлювати, а також керувати програмами в хмарі.
- Економічно вигідно.
- Висока масштабність.
- Один і той самий скрипт можна запускати на одному пристрої паралельно.

Недоліки хмарного мобільного тестування:

- Менше контролю.
- Немає такого високого рівня безпеки.
- Залежність від інтернет-з’єднання.

Онлайн тестування:

- API (application programming interface) – основний інтерфейс для взаємодії з іншими програмами.
- GUI (graphic user interface) – графічний інтерфейс, використовується для взаємодії з користувачем.
- Net (networking interface) – працює через мережу і використовується як просунутими користувачами, так і програмами.

2.3 Засоби автоматизації тестування мобільних додатків

Тести можуть використовувати всі ці інтерфейси для взаємодії з додатком. При ручному тестуванні посередником між тестами і додатком є тестувальник: він перетворює текст тест-кейсів на дієвий з одним з інтерфейсів програми.

Для автоматизації потрібно замінити тестувальника на інструменти, які вміють взаємодіяти з одним або декількома інтерфейсами програми [28]. Також будуть потрібні утиліти для запуску і формування набору тестів.

Разом всі ці інструменти називаються стеком автотестування. Щоб зрозуміти, як вони взаємодіють в стеці, необхідно їх класифікувати. Представлена класифікація умовна і необхідна в першу чергу для розуміння інструментів та їх поєднання .

Всього існує чотири групи інструментів: драйвери, надбудови, фреймворки і комбайни.

2.3.1 Драйвер

Утиліти автотестування, як і інші програми, можуть взаємодіяти з додатком тільки через програмний інтерфейс – по-іншому вони не вміють. Для роботи через інші інтерфейси існують спеціальні програми – драйвери.

Драйвер – програма, яка надає API для одного з інтерфейсів програми.

Для кожного інтерфейсу, крім, власне, API, необхідний свій драйвер. Наприклад, коли ви даєте драйверу для GUI команду “Натиснути на кнопку Menu”, він сприймає її через API і відсилає в тестований додаток, де ця команда перетворюється в клік по графічній кнопці Menu [24]. Для взаємодії з API додатка драйвери не потрібні або майже не потрібні – взаємодія програмна. А ось при роботі з іншими інтерфейсами без них не обійтися.

Найбільш складними зазвичай є драйвери для GUI, оскільки цей інтерфейс дуже відрізняється від звичайного для програми спілкування кодом. При цьому в автоматизованому тестуванні мобільних додатків GUI вважається найбільш

актуальним, тому в інтеграційному тестуванні використовувати найчастіше доводиться саме його. Найбільш популярні драйвера для GUI в мобільному тестуванні – UIAutomator і Espresso для Android, XCUITest – для iOS.

2.3.2 Надбудова

Коли функціоналу драйвера не вистачає або він незручний і складний, над ним з’являється ще один рівень, який я буду називати надбудовою [33].

Надбудова – програма, яка взаємодіє з додатком через один або кілька драйверів, підвищуючи зручність їх використання або розширюючи їх можливості.

У надбудови можуть бути наступні функції:

- Модифікація поведінки (без зміни API). Наприклад, додаткове протоколювання, валідація даних, очікування виконання дії протягом певного часу.
- Підвищення зручності і / або рівня абстракції API через використання синтаксичного цукру – зручних назв функцій, більш коротких звернень до них, уніфікованого стилю написання тестів; неявне управління драйвером, коли, наприклад, він ініціалізується автоматично, без необхідності прописувати кожну таку дію вручну; спрощення складних команд на кшталт вибору події з календаря або роботи зі списками, які прокручуються; реалізацію альтернативних стилів програмування, таких як процедурний стиль або fluent.
- Уніфікація API драйверів. Тут надбудова надає єдиний інтерфейс для роботи відразу з декількома драйверами. Це дозволяє, наприклад, використовувати один і той самий код для тестів на iOS і Android, як у популярній надбудові Appium.

2.3.3 Фреймворк

З іншого боку тестів знаходиться фреймворк запуску. В цій статті я буду коротко називати його “фреймворк”.

Фреймворк – це програма для формування, запуску і збору результатів запуску набору тестів.

До завдань фреймворка входять:

- Формування, угруповання і упорядкування набору тестів.
- Розпаралелювання набору (опціонально).
- Створення фікстур.
- Запуск тестів.
- Збір результатів їх виконання.
- Формування звітів про виконання (опціонально).

Можна помітити, що ці функції не пов’язані з тестуванням тільки мобільних додатків – їх можна успішно застосовувати і в тестуванні десктоп- і веб-додатків. Справа в тому, що фреймворк не повинен забезпечувати взаємодію тестів і додатки – він працює тільки з тестами, і тип програми не має значення [35].

Якщо драйвери і надбудови знаходяться між тестами і додатком, то фреймворк знаходиться над тестами, організовуючи їх запуск. Тому важливо не плутати поняття “драйвер” і “фреймворк”. Звичайно, в деяких фреймворках є власні драйвери для роботи з додатками, але це зовсім необов’язкова умова. Найпомітніші фреймворки в мобільному тестуванні – xUnit і Cucumber.

2.3.4 Комбайни

Нарешті, ще одна група утилітів, що використовуються для автоматизації тестування мобільних додатків, – це комбайни, які поєднують в собі і фреймворки, і драйвери (причому не тільки мобільні), і навіть можливості розробки.

Xamarin.UITest, Squish, Ranorex – всі вони підтримують автоматизацію тестування iOS-, Android-, веб-додатків, а два останніх – ще й десктоп-додатків [32].

Безсумнівно, реальний пристрій – найкраще рішення, для тестування мобільного додатку. Тестування на реальному пристрої завжди дає максимальну точність результатів. Насправді нелегко вибрати найбільш підходящий пристрій. Переваги для тестування мобільних додатків на реальних пристроях:

- Висока точність результату тестування.
- Проста реплікація помилок. Такі моменти, як: ємність батареї, геолокація, push-повідомлення, вбудовані датчики пристроїв, легко тестуються.
- Можливість перевірки вхідних переривань (дзвінків, SMS).
- Можливість тестування мобільного застосування в реальних умовах і умовах.
- Немає помилкових спрацьовувань.

Недоліки тестування мобільних додатків на реальних пристроях :

- Існує величезна кількість часто використовуваних пристроїв.
- Додаткові витрати на обслуговування пристроїв.
- Обмежений доступ до пристроїв, які часто використовуються в зарубіжних країнах.

Висновки до 2 розділу

З точки зору особливостей поставленої проблеми для визначення можливостей вирішення задачі прискорення тестування мобільних додатків було проведено аналіз існуючих методів. Також було проаналізовано недоліки та переваги існуючих засобів автоматизованого тестування.

На основі проведеного аналізу було обрано методи та інструментальні засоби для подальшого створення методики прискорення автоматизованого тестування мобільних додатків.

РОЗДІЛ 3 ІНСТРУМЕНТАЛЬНІ ЗАСОБИ МЕТОДИКИ ПРИСКОРЕННЯ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ МОБІЛЬНИХ ДОДАТКІВ

Тестування – один з найважливіших етапів контролю якості в процесі розробки програмного забезпечення. Автоматизоване тестування є його складовою частиною. Воно використовує програмні засоби для виконання тестів і перевірки результатів виконання, що допомагає скоротити час тестування і спростити його процес.

3.1 Основні підходи до автоматизації тестування

Існує два основних підходи [1] до автоматизації тестування:

- тестування на рівні коду
- тестування користувацького інтерфейсу (GUI-тестування).

До першого типу належить, зокрема, модульне тестування. До другого – імітація дій користувача за допомогою спеціальних тестових фреймворків. Найбільш поширеною формою автоматизації є тестування додатків через графічний користувальницький інтерфейс. В роботі розглянуто один з таких підходів.

Популярність такого виду тестування пояснюється двома факторами:

1. по-перше, додаток тестується тим же способом, яким його буде використовувати людина;
2. по-друге, можна тестувати додаток, не маючи при цьому доступу до вихідного коду.

Однією з головних проблем автоматизованого тестування є його трудомісткість, незважаючи на те, що воно дозволяє усунути частину рутинних операцій і прискорити виконання тестів, великі ресурси можуть витрачатися на оновлення самих тестів. Це відноситься до обох видів автоматизації. При рефакторінгу програм часто буває необхідно оновити і модульні тести, і зміна коду тестів може зайняти стільки ж часу, скільки і зміна основного коду. З іншого боку, при зміні інтерфейсу додатку необхідно заново переписати всі тести, які пов’язані з оновленими вікнами, що при великій кількості тестів може відняти значні ресурси.

Другою проблемою є час виконання автоматизованих тестів. Тести GUI на великих проектах зазвичай запускаються звечора, аби до ранку результат їх проходження був готовим. Це сповільнює вихід нових версій та не дає швидко виправити помилки, якщо вони раптом з’являються в продукті.

Ще зовсім недавно тестування програм проводилося вручну або самими програмістами, або користувачами, що навряд чи можна було назвати системним

підходом і до того ж це не дозволяло оцінювати якість коду. Трохи пізніше тестування виділилося в окрему галузь знань у складі розробки програмного забезпечення, але швидко прийшло розуміння того, що тестування вручну неефективне, оскільки вимагає великих трудових ресурсів і багато часу.

Перші засоби автоматизації тестування практично представляли собою бібліотеки, які можна було використовувати для написання тестів, що вимагало від тестувальника вміння програмувати на рівні розробника.

Сучасні засоби автоматизованого тестування дозволяють створювати автоматизовані тести з мінімальною участю людини (на основі дій користувача автоматично генерується сценарій тестування), проте вони не можуть гарантувати якість, яку дає метод програмування тестів.

3.2 Класифікація засобів та план тестування

Прийнято розділяти тестування за рівнями завдань і об’єктів на різних стадіях і етапах розробки програмного забезпечення [5]:

1. Функціональне тестування підсистем і програмного забезпечення в цілому з метою перевірки ступеня виконання функціональних вимог до програмного забезпечення – рекомендується проводити окремою групою тестувальників, які не підпорядковані керівнику розробки;

2. Модульне тестування частин програмного забезпечення (компонентів) з метою перевірки правильності реалізації алгоритмів – виконується розробниками;

3. Тестування навантаження (у тому числі стресове) для виявлення характеристик функціонування програмного забезпечення при зміні навантаження (інтенсивності звернень до нього, наповнення бази даних і тому подібне) – для виконання цієї роботи потрібні висококваліфіковані тестувальники і дорогі засоби автоматизації експериментів.

На ринку засобів автоматизованого тестування сьогодні представлено багато продуктів [5], більшість зібрана в таблиці 3.2. Найбільш потужними з них є: HP (QuickTest Professional, WinRunner), IBM (Robot, Functional Tester), AutomatedQA

(TestComplete) та Selenium, який зараз часто використовується для тестування Web-додатків. Частина з них використовує стандартні мови програмування (наприклад, QTP використовується в якості мови розробки скриптів VB, а Functional Tester, реалізований в середовищі Eclipse, дозволяє створювати скрипти на Java), а частина застосовує свої власні спеціалізовані мови (наприклад, Robot використовує мову SQABasic для написання скриптів).

Таблиця 3.2 – Засоби автоматизованого тестування

Назва	Розповсюдження, Ціна	Мова тестів	Технології
Функціональне тестування			
Canoo WebTest	Безкоштовний	HTML, Groovy	Web-технології
Codeception	Безкоштовний	PHP	Web-технології
Coded UI Test	Коштовний	C#	.NET 2.0, 3.0, 3.5, 4.
GUIdancer	Коштовний	Keyword-driven	Swing, SWT/RCP, GEF, HTML
HP QuickTest Professional	Коштовний, (~$10000)	VBScript	Web, Java, .Net, WPF, SAP
IBM Rational Functional Tester	Коштовний, $6000	Java, Visual Basic	Java,.NET,Win32,HTML,Terminal
M-eux	Коштовний	Java, C#, VBScript	Java, C#, VBScript
Ranorex	Коштовний, 290-1190 EUR	C#, VB.NET, Python (IronPython)	.NET (C#, VB.NET), WPF (XAML
Robot Framework	Безкоштовний	Python, Java	Java, .NET
RoutineBot	Коштовний, $495	Pascal, JavaScript, Basic	Windows Forms, Flex
Selenium	Безкоштовний	HTML, Java, C#, Perl, PHP, Python, Ruby	DHML, JavaScript, Ajax
Sikuli	Open Source	Sikuli Script	Jython, Windows, Linux, MacOS
SWAPY	Open Source	Python	Windows, Python
T-Plan Robot	Коштовний	Proprietary language Java	VNC, Windows, Linux, Java, C
TestComplete	Коштовний, $1000-5000	VBScript, JScript, DelphiScript, C++Script, C#Script	IE, Firefox, Chrome, Flash,
Testdroid	Коштовний,$595	Java	Java
Testing Anywhere	Коштовний, $7000	візуальне проектування	VB.NET, C#, C++, Win32, VB6
TestPlan	Безкоштовний	власна скрипкова мова	HTML, DHTML, JavaScript
Twist	Коштовний	Java, Groovy	Selenium, Java, Swing, Groovy.
Watir	Open Source	Ruby, Java, .NET, Perl	HTML, JavaScript
Модульне тестування			
Codeception	Безкоштовний	PHP	Web -технології
Selenium	Безкоштовний	HTML, Java, C#, Perl, PHP, Python, Ruby	DHML, JavaScript, Ajax
Testing Anywhere	Коштовний, $7000	візуальне проектування	VB.NET, C#, C++, Win32, VB6
Тестування навантаження			
AgileLoad	Коштовний, $70	SCL	Web -технології
Apache JMeter	Open Source	визуальное проектирование	HTTP, HTTPS, SOAP, JDBC, LDAP
BrowserMob	Коштовний	Selenium Record&Playback	AJAX, Flash
Testing Anywhere	Коштовний, $7000	візуальне проектування	VB.NET, C#, C++, Win32, VB6

Більшість інструментів орієнтовані на роботу з Web-додатками або зі звичайними додатками, написаними із застосуванням технологій Net або Java. Розробники засобів автоматизованого функціонального тестування досить оперативно реагують на появу нових механізмів і платформ розробки програмного забезпечення. Незважаючи на те, що на ринку існує величезна різноманітність різних продуктів для автоматизованого тестування, деякі компанії-розробники програмного забезпечення створюють власні інструменти, пристосовані для тестування розроблюваних ними додатків. Причинами цього є висока вартість засобів автоматизованого тестування та унікальність програмного забезпечення, що тестується, яка не дозволяє використовувати стандартні засоби автоматизації тестування. Крім засобів тестування існують так звані засоби підтримки процесу тестування, що дозволяють вести облік вимоги і тест-кейси, проводити аналіз покриття вимоги тестами, керувати ходом виконання тестування, вести облік виявлених дефектів і тому подібне. Лідирує в даній області Web-додаток HP Quality Center – іструмент управління процесом тестування, який, включаючи управління вимогами і дефект-менеджмент, інтегрований із засобами функціонального і навантажувального тестування HP QuickTest Professional.

Важливим моментом при створенні тестових систем та написанні тестів є наявність плану тестування, що визначений міжнародним стандартом IEEE 829-1983 [6].

В ньому повинні бути наступні розділи:

- що буде тестуватися (тестові вимоги, варіанти тестів);
- якими методами буде тестуватися система та критерії, за якими буде визначатися успіх тестів;
- план робіт та ресурси (тестувальники, техніка).

Також слід визначити певні критерії вдалого/невдалого завершення тестів, ризики і плани керування та конфігурації середовища та керування та т.п. Коли зрозуміло, що і навіщо потрібно тестувати, і є план дій, саме час задуматися про те, як це зробити ефективніше, швидше і якісніше. Сучасне програмне забезпечення – це складний об’єкт, і вручну з ним справлятися важко і дорого. До того ж при

«ручному» тестуванні результати кожного виконання тестів пропадають, і їх важко повторити. Для того щоб збільшити обсяг перевірок і підвищити якість тестування, забезпечити можливість повторного використання тестів при внесенні змін у програмне забезпечення застосовують засоби автоматизації тестування.

Сьогодні Україна відстає від решти комп'ютерного світу щодо застосування засобів автоматизації тестування, і для цього є декілька причин:

1. Неправильне ставлення до тестування як такого – багато керівників вважають, що розробник може написати програму, яка не містить помилок.

2. Висока вартість інструментів автоматизації тестування.

3. Бажання заощадити на кваліфікованих кадрах – робота спеціаліста з засобів автоматизації тестування обходиться дорожче, ніж праця звичайного тестувальника.

4. Обмеження по термінах – автоматизація тестування вимагає значних тимчасових витрат і має сенс тільки в тому випадку, якщо проект як мінімум середньостроковий.

5. Невдалий досвід застосування таких засобів і очікування миттєвого ефекту від їх впровадження. Адже результати впровадження таких продуктів помітні далеко не відразу і витрати на автоматизацію окупаються за тривалий період часу, що не дозволяє повністю відмовитися від тестування вручну.

Для розробки автоматизованих тестів та прискорення їх тестування серед вищезазначених було обрано наступні засоби:

- Java – мова програмування для написання тестів
- Appium – драйвер для запуску тестів
- TestNG – тестовий фреймворк
- Allure – фреймворк для генерації результатів автоматизованого тестування

Саме прискорення виконання автоматизованих тестів буде досягнуто за рахунок використання багатопотокового виконання тестів, що досягнуто за рахунок використання мови програмування Java, попередньої ініціалізації всіх

графічних елементів та використанню розділення кроків при виконанні тестових сценаріїв (для цього буде використано фреймворк TestNG).

Засоби, що використані для прискорення автоматизованого тестування у даній роботі, описані детальніше у наступних підрозділах.

3.3 Java

Мова програмування Java зародилася в 1991р. в лабораторіях компанії Sun Microsystems inc. Як не дивно, поштовхом для створення Java стала зовсім не Internet. Головним мотивом була потреба в мові програмування, яка не залежала б від платформи (тобто від архітектури) і яку можна було б використовувати для створення програмного забезпечення, яке вбудовується в різноманітні побутові електронні прилади, такі як мобільні засоби зв'язку, пристрої дистанційного управління тощо. Розробка першої робочої версії зайняла 18 місяців і вона мала назву "Oak", але 1995 р. проект був перейменований на "Java".

Період становлення Java співпав за часом з розквітом міжнародної інформаційної служби World Wide Web. Ця обставина відіграла вирішальну роль в майбутньому Java, оскільки Web теж вимагала переносних програм. Як наслідок, були зміщені акценти в розробці Sun з побутової електроніки на програмування для Internet.

3.3.1 Безпечність

World Wide Web висунула Java на передній край програмування, і Java, в свою чергу, сильно вплинула і навіть змінила обличчя Internet, розширивши спектр об'єктів, які можуть розповсюджуватись у кіберпросторі. Програми нової форми – аплети – завантажуються з віддаленого сервера і можуть запускатися динамічно, тобто без участі користувача. До появи Java такий підхід був неприпустимий з

міркувань безпеки та переносимості. В архітектурі аплетів зроблено ряд штучних обмежень, які роблять їх цілком безпечними. Перш за все, Java є інтерпретованою мовою і простір ресурсів Java-програми обмежений так званою віртуальною Java-машиною (VJM), яка може контролювати поведінку програми і захищати систему від побічних ефектів, які можуть виникати з вини аплета. Крім того, в мові Java є додаткові обмеження, які не дозволять аплету стати "троянським конем". Зокрема, Java-аплет не може отримати доступ до локального жорсткого диску. При такій спробі генерується виключна ситуація.

3.3.2 Ефективність

Оскільки аплети Java інтерпретуються, а не компілюються, то їх виконання на різних платформах значно полегшується. В цьому випадку достатньо створити для кожної платформи виконуючу Java-систему. Якщо існує така система для даної операційної системи, то будь-яка Java-програма може виконуватись в даному середовищі без додаткової компіляції на цій платформі. Проте Java не є інтерпретованою мовою в чистому розумінні. Програма на Java компілюється. Результатом роботи компілятора Java є байткод (bytecode). Байткод – це оптимізований набір команд, призначений для виконання уявним пристроєм – віртуальною Java-машиною. В такий спосіб витрати на інтерпретацію зводяться до мінімуму, оскільки байкод вже є оптимізованим, і досягається досить висока продуктивність Java-програм. Наведені вище особливості дають підставу розглядати Java не як ще одну мову програмування, а як окрему інформаційну технологію. Таким чином, інтерпретація – це найлегший шлях до перенесення програм, реалізований в Java технології. Незважаючи на те, що мова Java була розроблена в розрахунку на інтерпретацію, технічно немає нічого такого, що б перешкоджало компіляції байткоду в виконуваний код. До байткоду, який пересилається по мережі, застосовується динамічна компіляція, але це ніяк не впливає на переносимість та безпеку, оскільки роботу програми все ще контролює

виконуюча система. Такий підхід застосовано в багатьох виконуючих системах Java, що забезпечує продуктивність на рівні оптимізованого коду C++.

Мова Java є однією з наймолодших в сімействі мов програмуваня і була розроблена з розрахунку на те, щоб професійний програміст міг легко її опанувати та ефективно використовувати. За основу Java взятий синтаксис C++ – безсумнівно однієї з найбільш популярних мов програмування сучасності. Проте, Java – це цілком самостійна мова програмування, і при її створенні не йшлося про будь-яку сумісність з C++. Тому деякі механізми реалізовані в Java інакше, а деякі зовсім відсутні. Ідеологічно ж Java побудована дещо інакше ніж C++. Розробники Java грунтувалися на досвіді розробки програм на C++ і прагнули позбутися можливостей, які зарекомендували себе непевними. Так, в Java відсутня перегрузка операторів а також автоматичне приведення несумісних типів – конструкції, які при неуважному використанні є джерелом важких для виявлення помилок. Взагалі, інтерфейси Java більш прості та прозорі для розуміння. Написати на Java програму з графічним інтерфейсом значно легше. Звичайно, простота інтерфейсів компенсується меншою гнучкістю, бібліотека Java не така багата, як стандартні бібліотеки C/C++. Але згадаймо, що Java задуманий для використання на різних платформах і тому реалізує в собі найбільш стандартні можливості задля легшої адаптації під конкретне середовище.

3.3.3 Об'єктно-орієнтована спрямованість

Від C++ Java успадкувала потужний механізм об'єктно-орієнтованого програмування. Оскільки Java розроблювався "на пустому місці", тобто не було потреби забезпечувати сумісність з попередніми версіями, розробники мали повну свободу мислення. В результаті був сформований ясний і прагматичний підхід до об'єктів. Вільно переймаючи ідеї, які реалізовувалися протягом останніх десятирічь, мові Java вдалося знайти рівновагу між парадигмою "все є об'єктом" і прагматичним підходом. Об'єктна модель Java проста і легко розширюється, в той

час як просі типи, як цілі, зберігаються як дані, що не є об'єктами, що дозволяє значно підвищити швидкість при їх обробці.

3.3.4 Стійкість до помилок

Багатоплатформеність середовища Web висуває надзвичайно високі вимоги до надійності програм. Як наслідок, при розробці Java приорітет був відданий можливості створення стійких до помилок програм. Java звільняє програміста від хвилювань з приводу багатьох поширених причин, які викликають помилки програмування. Як вже згадувалося, Java є строго типіованою мовою програмування. Ще виконуюча система Java бере на себе "прибирання сміття", тобто автоматично звільняє пам'ять, яка була розподілена динамічно. Звичайно, це дещо знижує ефективність коду, але запобігає типових помилок, коли програміст забуває звільнити виділену пам'ять, або, навпаки, звільняє пам'ять, яка ще використовується. Java підтримує об'єктно-орієнтовану обробку виключних ситуацій подібно до C++. Але на відміну від C++ в Java обробка виключних ситуацій є обов'язкувою. Тобто неможливо скомпілювати програму, яка відкриває файл, не обробивши можливі помилки типу "файл не знайдено", які виникають при цьому. Добре написана Java-програма може сама обробляти всі помилки часу виконання.

3.3.5 Підтримка багатопоточності

Java розробляялася з орієнтацією на вимоги до створення інтерактивних програм, які працюють з мережею. З цією метою Java підтримує багатопоточне програмування, яке дозволяє легко розробляти програми, що викинують багато процесів одночасно. Виконання Java-програми засновано на елгантному, але в той

самий час високоорганізованому рішенні багатопроцесової синхронізації, яке дозволяє вам створювати високоефективні інтерактивні системи.

3.3.6 Незалежність від архітектури

Основним питанням для розробників Java стало питання довготривалості та переносимості. Одна з головних проблем, із якою зустрілися програмісти, полягала в відсутності гарантій того, що написана сьогодні програма завтра працюватиме з тим же успіхом, причому на тій самій машині. Оновлення операційної системи, модернізація процесора та зміна об'єму оперативної пам'яті можуть призвести до збою програми. Рощзробники Java, прагнули змінити цю ситуацію і прийняли декілька важких рішень відносно мови Java та процесу виконання Java-програми. Їх мета полягала в тому, щоб "одного разу написане працювало всюди, в любий час і завжди". Внаслідок цього Java є системою, яка легко розширюється за рахунок створення нових стандартних класів та бібліотек.

3.3.7 Розподіленість

Мова Java призначена для створення програм, які працюють в розподіленому середовищі Internet на базі протоколів TCP/IP. Насправді доступ до ресурсів за допомогою URL відрізняється від доступу к файлу. Крім того в Java наявний засіб передачі повідомлень в межах внутрішнього адресного простору. Це дозволяє забеспечити віддалене виконання процедур. Ці інтерфкйси включені у пакет RMI (remote metod invocation). Цей засіб привносить високий рівень абстракції в програмування дл я середовища клієнт/сервер.

Java-програми несуть у собі значний обсяг інформації про типи часу виконання (run-time type information), яка використовується для дозволу доступу до об'єктів під час роботи програми. Це дозволяє забезпечити безпечну та оптимальну

динамічну компоновку. В такий спосіб досягається захищеність середовища виконання аплетів.

3.4 Appium

Провівши аналіз доступних драйверів для тестування мобільних додатків, для подальшої роботи було обрано Appium Driver. Порівняльну характеристика драйверів наведено в таблиці 3.1.

Таблиця 3.1 – Порівняльна характеристика драйверів для тестування мобільних додатків

	підтримка Android	підтримка iOS	написання тестів на Java	зручність використання	можливість роботи з .apk \ .app файлами
UIAutomator	+	-	-	+	-
Espresso	+	-	+	-	-
XCUITest	-	+	-	-	-
Appium	+	+	+	+	+

Appium – відомий інструмент автоматизації мобільних додатків (Android & iOS), з можливістю написання тестів на великій кількості мов програмування / фреймворками, серед яких є Java.

Великою перевагою Appium є те, що він дає доступ до тестування одночасно різних додатків та браузерів на додатку, тобто при наявності в мобільному додатку

зовнішніх джерел (наприклад, логін та реєстрація через Facebook або Gmail) є можливість доступу та, як наслідку, покриття автоматизованими тестами і цього функціоналу.

Appium дозволяє створювати та зберігати безліч варіантів девайсів для запуску автоматизованого тестування (рисунок 3.4).

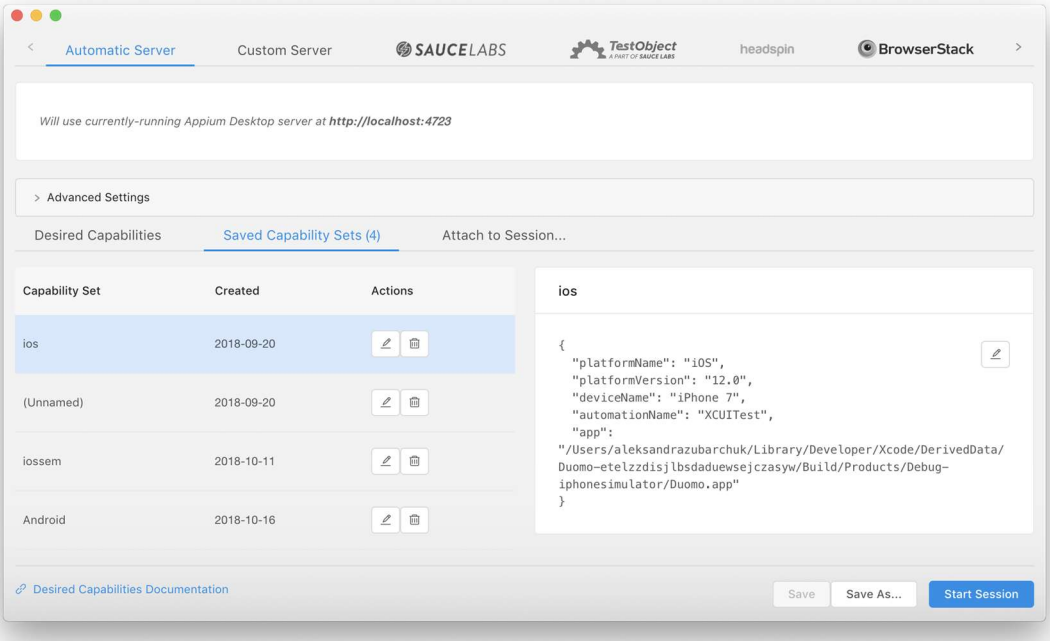


Рисунок 3.4 – Збереження мобільних девайсів для запуску автоматизованих тестів у драйвері Appium

Позитивні особливості Appium:

1. Порівняно легко встановити, налаштувати, запустити перший тест.
2. Підтримується Android & iOS.
3. Багато способів ідентифікації елементів.
4. Синтаксис дуже схожий на Selenium, завдяки чому можна дуже швидко почати писати тести, не треба вивчати новий API.
5. Дозволяє писати DeepTests (тобто тести за структурою поточної Activity та ініціалізації доступних на ній елементів, з якими треба працювати і змінюють список перевірок в залежності від того що зараз представлено на екрані – а не просто діють по строго певних кроків).

Негативні особливості Appium:

1. Не завжди знаходяться елементи з першого разу, а якщо елемент вже знайдено не завжди є можливість читати властивості знайдених елементів. Даний недолік компенсується написанням функцій обгорток.

2. Немає можливості розпізнавати елементи по картинці – тобто для автоматизації мобільних ігор Appium не підходить.

3.5 Фреймворк testNG

Було проведено аналіз наявних фреймворків для тестування мобільних додатків, для подальшої роботи було обрано testNG. Порівняльну характеристика фреймворків наведено в таблиці 3.2.

Таблиця 3.2 – Порівняльна характеристика фреймворків автоматизованого тестування

	Driver	Мова програмування	Підтримка анотації кроків	Параметризація
XUnit	Web Driver	Java / C#	+	-
Cucumber	Web Driver	PHP	-	-
TestNG	Selenium / Appium Driver	Java	+	+
JUnit	Selenium	Java	-	+

TestNG – це тестова система для мови програмування Java, створена Cédric Beust і натхненна JUnit та NUnit. Метою проекту TestNG є охоплення більш широкого кола тестових категорій: одиничний, функціональний, кінцевий, інтеграційний та інші, з більш потужними та простими функціональними можливостями.

Основні функції TestNG:

- Підтримка анотацій.
- Підтримка параметризованих та керованих даних для тестування (з @DataProvider та / або конфігурацією XML).
- Підтримка декількох екземплярів одного класу тесту (з @Factory)
- Гнучка модель виконання. TestNG може запускатися або через Ant через build.xml (з певним тестовим набором або без нього), або через плагін IDE з візуальними результатами.
- Паралельне тестування: виконання тестів у великих групах потоків з різними доступними політиками (всі методи у власній течії, одна гілка на тестовий клас і т. д.), що прискорює виконання тестів.
- Типові функції JDK для виконання та реєстрації (без залежності).
- Розподілений тест: дозволяє розподілити тести на машинах, що їх виконують.

TestNG підтримується, нестандартно або через плагіни, кожним з трьох основних IDE Java – Eclipse, IntelliJ IDEA та NetBeans. Він також постачається зі спеціальним завданням для Apache Ant і підтримується системою збирання Maven. Сервер безперервної інтеграції Hudson має вбудовану підтримку для TestNG і може відслідковувати та аналізувати результати тестування з часом. Більшість інструментів охоплення коду Java, такі як Cobertura, працюють без проблем з TestNG.

3.6 Allure Report

Є багато цікавих тестових фреймворків для різних мов програмування. На жаль, лише деякі з них можуть забезпечити гарне представлення результатів тестового виконання. Група тестування "Яндекс" працює над Allure – платформою з відкритим кодом, розроблений для створення звітів про виконання тестів, які є зрозумілими для кожного, хто їх перегляне.

Allure базується на стандартних результатах xUnit, але додає деякі додаткові дані. Будь-який звіт створюється в два етапи. Під час виконання тесту (перший крок) невелика бібліотека під назвою адаптер, прикріплена до тестової конфігурації, зберігає інформацію про виконувані тести у файлах XML. Адаптери доступні для популярних тестових платформ Java, PHP, Ruby, Python, Scala та C #.

Під час генерації звітів (другий етап) файли XML перетворюються у звіт HTML. Це можна зробити за допомогою інструмента командного рядка, плагіна для CI або інструмента створення (рисунок 3.6).

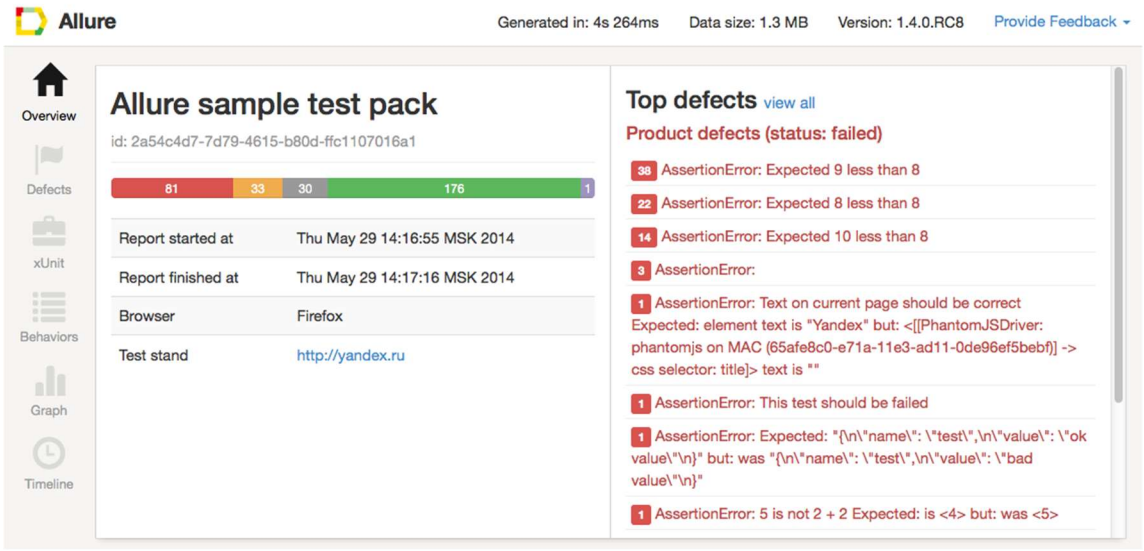


Рисунок 3.6 – генерація звіту про виконане тестування в Allure

Створена методика прискорення автоматизованого тестування мобільних додатків передбачає використання усіх вищезазначених інструментальних засобів.

Загалом, методика являє собою сукупність кроків, які необхідно виконати, аби проект з розроблюваними автоматизованими тестами виконувався швидше.

Методика передбачає:

1. Створення сценаріїв тестування мобільного додатку.
2. Пошук елементів сторінок у мобільному додатку.
3. Опис знайдених елементів в проекті тестування.
4. Групування елементів в окремі класи – об’єкти сторінок.
5. Використання анотацій кроків при програмуванні дій тестів.
6. Попередня ініціалізація об’єктів сторінок безпосередньо перед початком тесту або тестового сценарію.
7. Використання багато-потоковості при запуску автоматизованих тестів.
8. Запуск автоматизованих тестів на виділеному сервері.

Схема послідовності кроків, необхідних для досягнення швидкодії тестів зображено на рисунку 3.6.

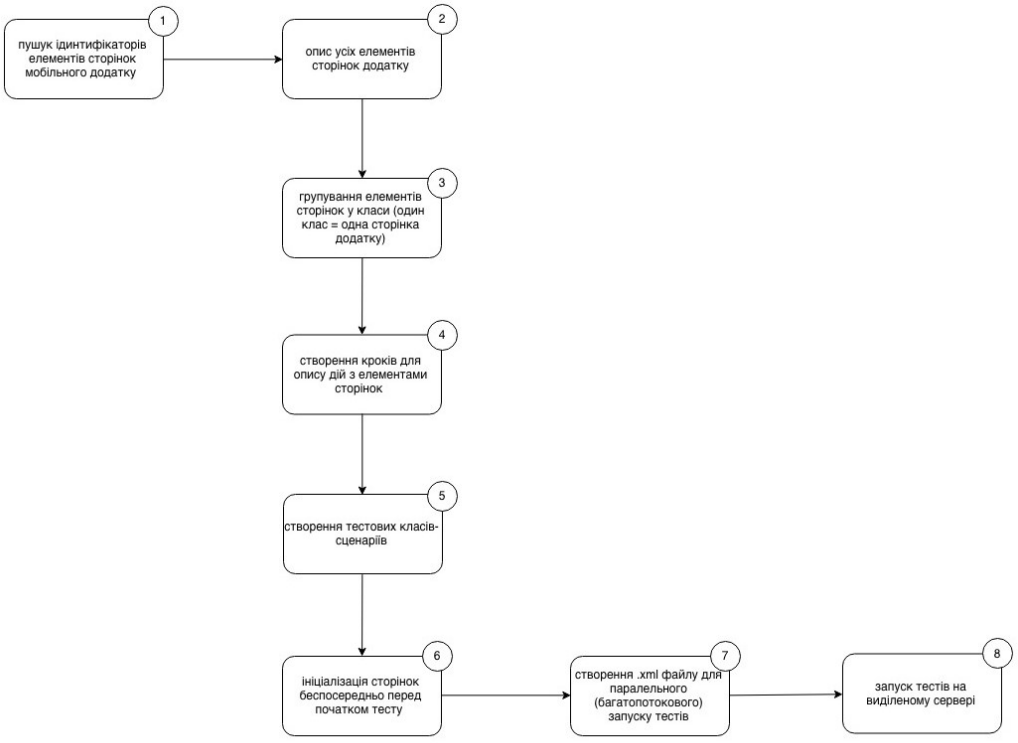


Рисунок 3.6 – Методика прискорення автоматизованого тестування мобільних додатків

Висновки до 3 розділу

Розроблено методику за допомогою якої можна досягти значного прискорення проведення автоматизованого тестування мобільних додатків. Представлено детальний опис методики та інструментальних засобів, необхідних для її використання. Методика передбачає обов’язкове виконання восьми кроків, додатково пропонується використовувати фрейморк для генерації автоматичного звіту про проведене тестування «Allure Report».

Методика може бути використана при тестуванні будь-яких мобільних додатків.

РОЗДІЛ 4 РОЗРОБКА СИСТЕМИ АВТОМАТИЗОВАНОГО ТЕСТУВАННЯ НА ОСНОВІ СТВОРЕНОЇ МЕТОДИКИ

Методику прискорення автоматизованого тестування мобільних додатків описано в попередньому розділі. Для практичної перевірки достовірності створеної методики, було розроблено систему тестування мобільного додатку “Duomo” з її використанням.

4.1 Опис та групування елементів сторінок мобільного додатку

Для доступу та можливості взаємодії з окремими елементами на сторінках мобільного додатку необхідно знати їх id (індивідуальний ідентифікатор елементу в обраному додатку) або XPath (або XML Path Language — це мова запитів, для вибору вузлів з XML документів).

Для того, аби отримати ці ідентифікатори є декілька способів:

1. Через інструмент розробника (через Android Studio для Android додатків або XCode для iOS додатків).
2. Через Appium inspector із відкритим файлом .apk, чи .app відповідно.

Було використано пошук елементів через Appium Inspector з наступних причин:

- не потрібно розбиратися в коді розробника, шукати файли окремих сторінок (рис. 4.1.1);

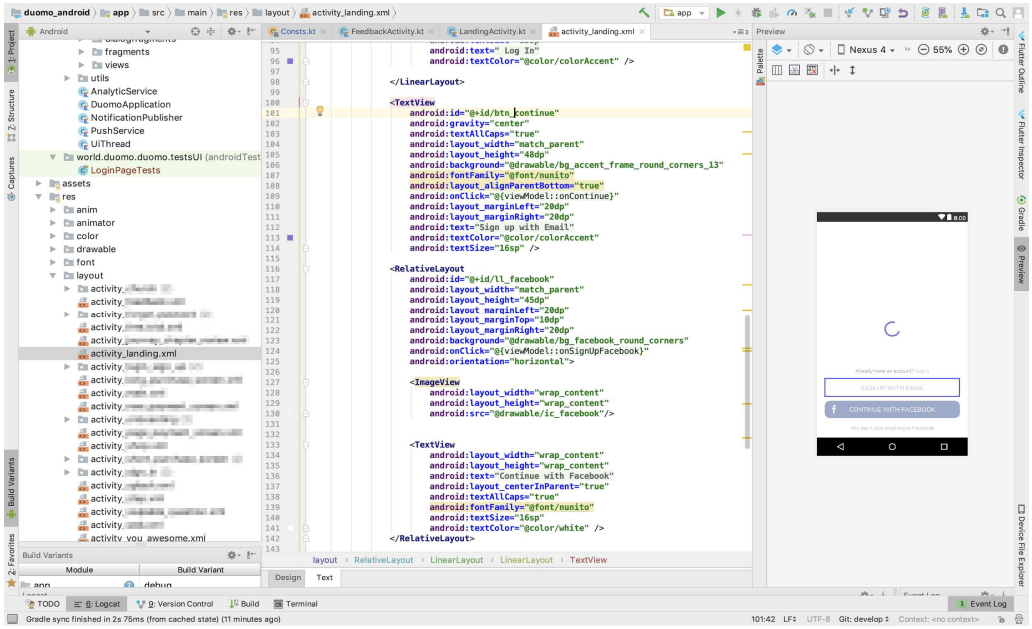


Рисунок 4.1.1 – Пошук елементів через інструмент розробника на прикладі Android Studio

- візуально видно, на якій саме сторінці додатку обираються елементи в даний момент, та при кліку на певний елемент видно усі атрибути обраного елементу (рис. 4.1.2);

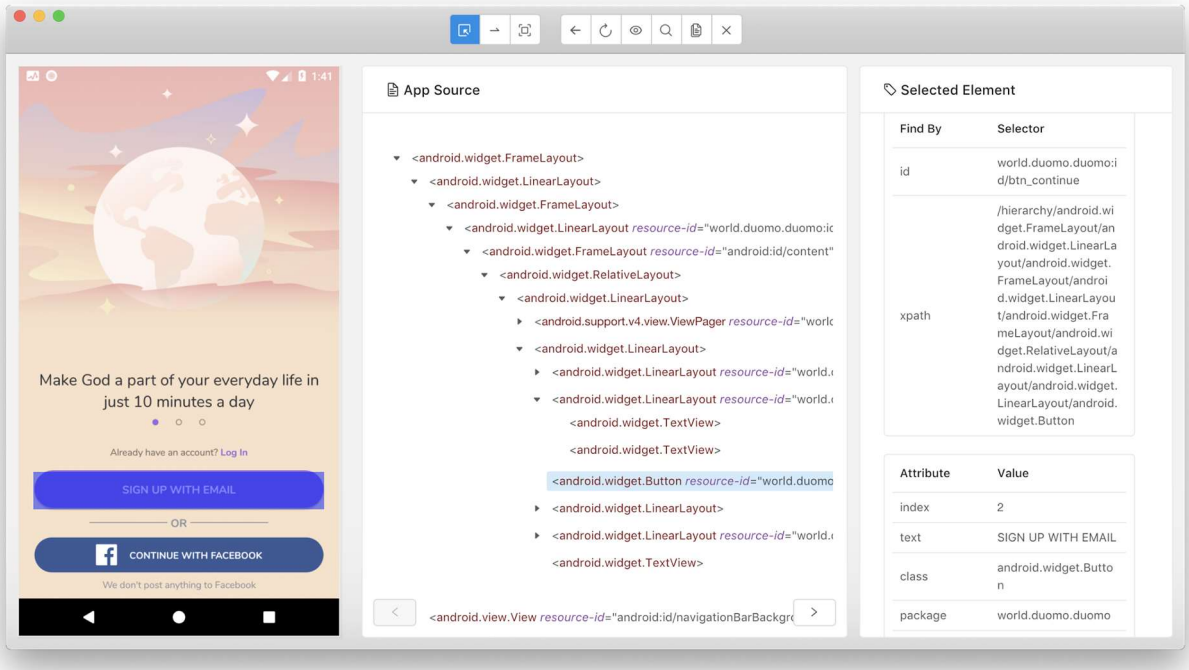


Рисунок 4.1.2 – Опис атрибутів та ідентифікаторів обраного елементу в Appium Inspector

- якщо розробником додатку не було присвоєно ідентифікатор певного елементу, Appium Inspector визначає XPath елементу (рис 4.2.3). Використовуючи пошук елементів через інструмент розробника, в разі відсутності ідентифікатору елемент залишається невизначеним.

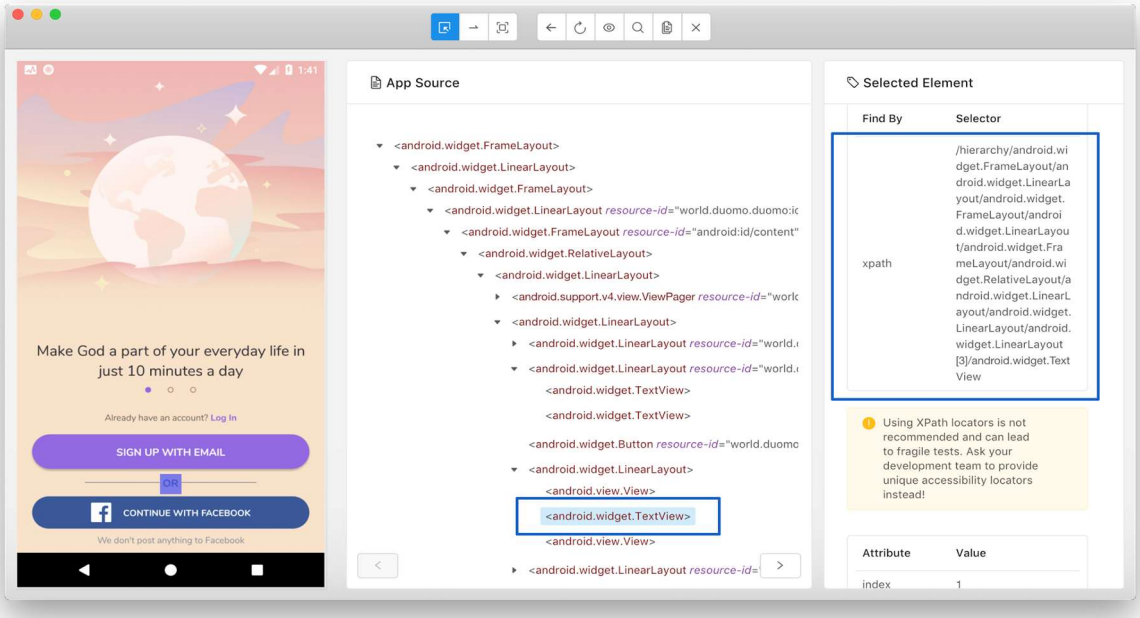


Рисунок 4.1.3 – Визначення XPath елемента для якого відсутній ідентифікатор у Appium Inspector

Після знаходження елементів одним із вищезазначених способів елементи треба описати в засобі створення автоматизованих тестів. Інтегрованим засобом розробки мною було обрано IntelliJ IDEA. Створюючи Maven Project в проекті автоматично генеруються папки. В папці src є дві вкладені папки: main та test, за назвами яких інтуїтивно зрозуміло, що test – слугує для створення вже готових тестів, а main – для опису усіх методів, констант, опису елементів, виклику драйверу.

Оскільки мобільний додаток, на прикладі якого створюються автоматизовані тести має версії на двох платформах, я винесла дві окремі папки: Android та iOS, в яких окремо для кожної платформи описано запуск драйверу, об’єкти сторінок та кроки тестів (рис. 4.1.4).

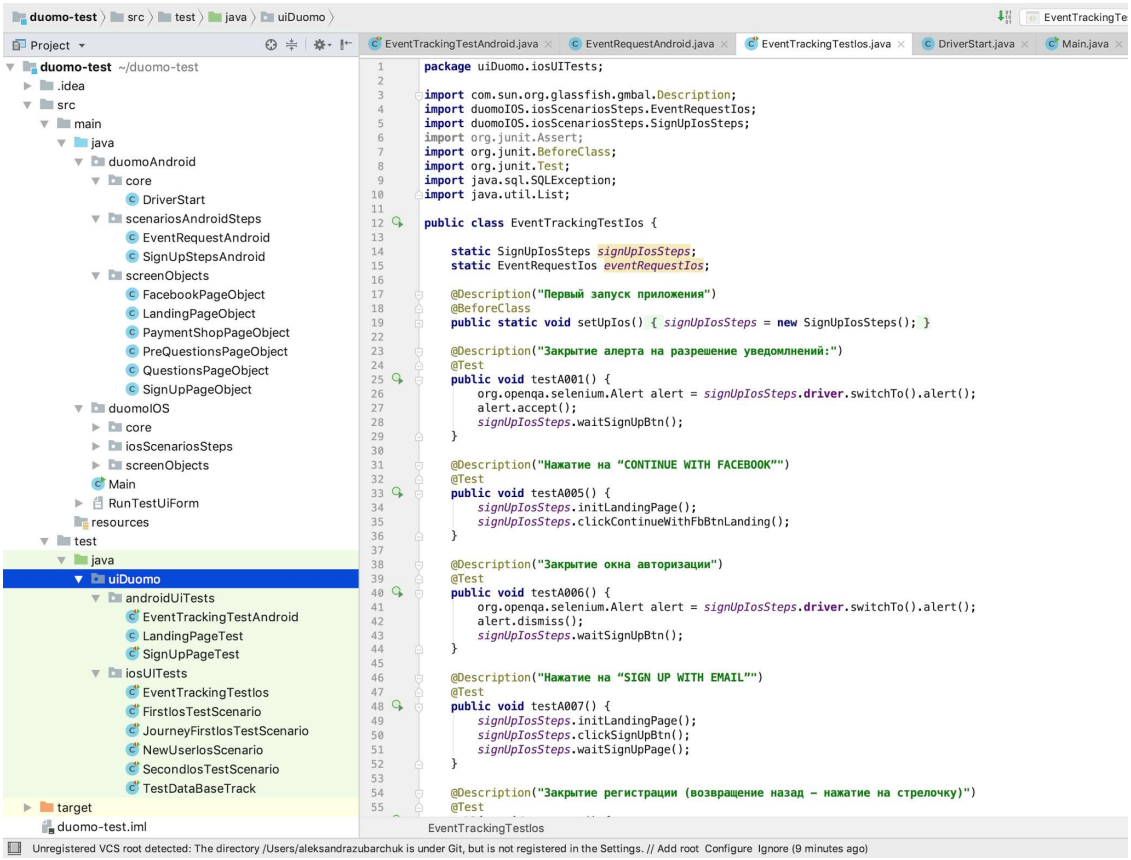


Рисунок 4.1.4 – Структуризація проекту з автоматизованими тестами для мобільного додатку “Duomo”

Для опису елементів кожної окремої сторінки створюється клас, в конструкторі якого кожному елементу сторінки надається ідентифікатор, що відповідає даному елементу додатку (рис. 4.1.5), далі, при написанні тестів, сторінка ініціалізується, на етапі відкриття. Ініціалізація сторінки потрібна для того, щоб тест при запуску мав доступ до всіх її елементів, а не шукав елемент при кожному його виклику.

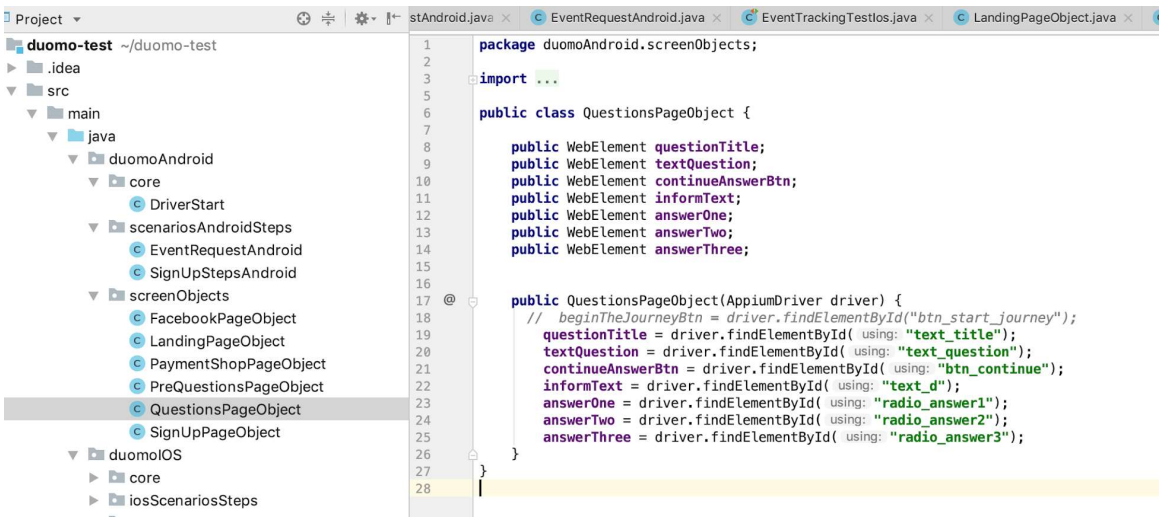


Рисунок 4.1.5 – Опис елементів сторінки мобільного додатку

4.2 Створення кроків виконання сценаріїв тестування мобільного додатку

Тестовий випадок/ситуація (Тест Кейс /Test Case) - це артефакт, що описує сукупність кроків, конкретних умов та параметрів, необхідних для перевірки реалізації функції, що тестується чи її частини.

Під тест кейсом мається на увазі наступна структура: Action > Expected Result > Test Result.

Створення методів для кроків тесту (Actions) в окремих класах необхідно, в першу чергу, для того, аби сам авто-тест виглядав структуризовано та мав чіткий перелік дій (тобто описував сукупність кроків). Також це необхідно для того, аби в разі падіння тесту можна було одразу зрозуміти, на якому кроці та за яких умов щось сталось.

Наприклад, один і той самий тест, написаний без попереднього створення методів-кроків, буде виглядати так (рис. 4.2.1):

```
@Test
public void testRegistration() {
SignUpIosPageObject signUpPageObject = new SignUpPageObject(AppiumDriver);
  signUpPageObject.nameLine.click();
  signUpPageObject.nameLine.sendKeys(name);
  signUpPageObject.emailLine.click();
  signUpPageObject.emailLine.sendKeys(emailAuto);
  signUpPageObject.passLine.click();
  signUpPageObject.passLine.sendKeys(passAuto);
  signUpPageObject.signUpBtn.click();
driverWait.until(ExpectedConditions.visibilityOfElementLocated(By.id("text_intro")));
  Assert.assertTrue(signUpScenario.isPreQuestionPageDisplayed());
}
```

Рисунок 4.2.1 - Приклад тестового кейсу без попереднього створення методів опису тестових кроків

А тест, з попереднім створенням тестових кроків - так (рис. 4.2.2):

```
@Test
public void testRegistration() {
  signUpScenario.initSignUpPage();
  signUpScenario.inputName(nameAuto);
  signUpScenario.inputEmail(emailAuto);
  signUpScenario.inputPass(nameAuto);
  signUpScenario.confirmSignUpBtn();
  signUpScenario.waitPreQuestionPage();
  Assert.assertTrue(signUpScenario.isPreQuestionPageDisplayed());
}
```

Рисунок 4.2.2 - Приклад тестового кейсу з попереднім створенням методів опису тестових кроків

Для опису кроків створюється окремий клас (для кожного сценарію окремий), в якому створюються методи (назва методу - назва кроку) з описом машинних дій, які необхідно зробити для того, аби виконати крок (рис. 4.2.3).

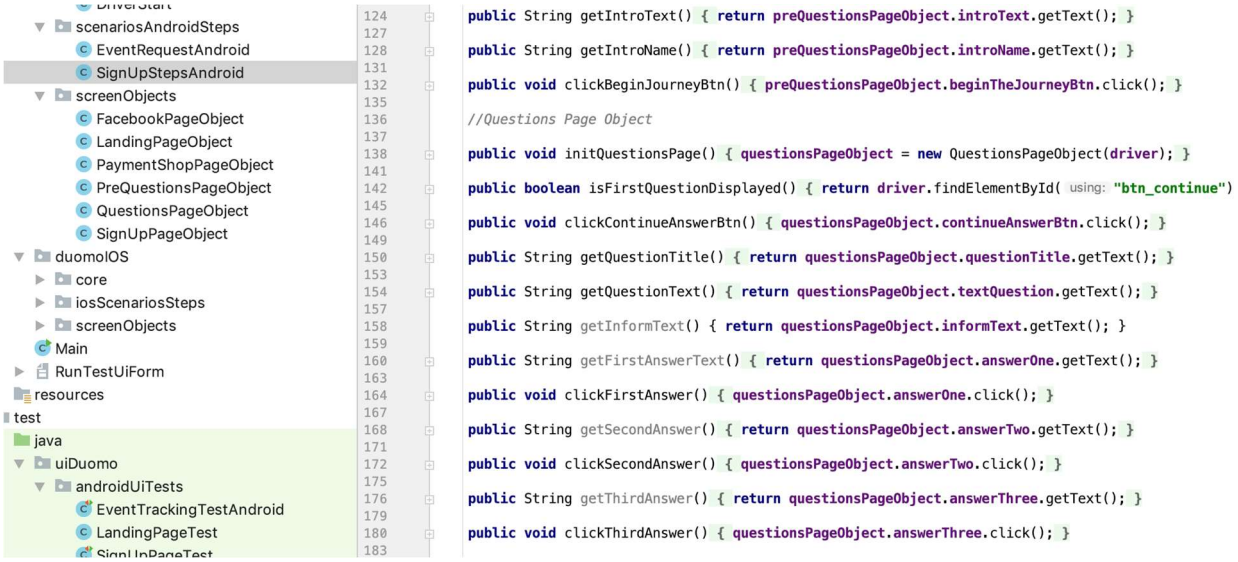


Рисунок 4.2.3 - Методи попереднього опису кроків

4.3 Ініціалізація сторінок додатку перед виконанням тесту

Ініціалізація — ряд дій, що передують виконанню програми, зокрема, встановлення програмних змінних в нуль, або надання їм інших початкових значень.

Ініціалізація об'єкта відбувається у методі, що називається конструктором (рис 4.3.1).

```
package duomoAndroid.screenObjects;

import ...

public class LandingPageObject {

    public WebElement signUpBtn;
    public WebElement logInLink;
    public WebElement continueWithFb;
    public WebElement textView;

    public LandingPageObject(AppiumDriver driver) {
        signUpBtn = driver.findElementById( using: "btn_continue");
        logInLink = driver.findElementById( using: "ll_signin");
        continueWithFb = driver.findElementById( using: "ll_facebook");
        textView = driver.findElementById( using: "text_view");
    }

}
```

Рисунок 4.3.1 - Конструктор класу Landing Page Object

Далі, при описі кроків тесту, створюється об’єкт потрібної сторінки, за рахунок чого всі елементи на сторінці, що описані в конструкторі, знаходяться в один крок (рис. 4.3.2).

```
public class SignUpStepsAndroid extends DriverStart {

    SignUpPageObject signUpPageObject;
    LandingPageObject landingPageObject;
    PreQuestionsPageObject preQuestionsPageObject;
    QuestionsPageObject questionsPageObject;
    FacebookPageObject facebookPageObject;
    PaymentShopPageObject paymentShopPageObject;

    //Landing Page

    public void initLandingPageAndroid() {
        landingPageObject = new LandingPageObject(driver);
    }

    public void waitLandingLoad() {
        driverWait.until(ExpectedConditions.visibilityOfElementLocated(By.id("btn_continue")))
    }

    public boolean isLandingPageDisplayed() {
        return driver.findElementById( using: "world.duomo.duomo:id/ll_signin")
                != null;
    }
```

Рисунок 4.3.2 - Створення об’єкту сторінки додатку при описі тестового сценарію

На цьому етапі стає зрозуміло, чому попередня ініціалізація сторінок пришвидшує виконання автоматизованого тесту. Наприклад, для кроку “ввести email”, драйверу необхідно:

1. Знайти поле вводу емейлу.
2. Натиснути (клыкнути) на поле вводу.
3. Очистити поле, якщо воно не було порожнім.
4. Ввести електронну адресу.

За наявності попередньої ініціалізації елементів сторінок пункт “знайти поле вводу електронної пошти” стає неактуальним, адже всі елементи на даній сторінці були знайдені раніше. Як видно на прикладах в таблиці 4.2.1 та таблиці 4.2.2, один тест кейс містить п’ять кроків, на початку яких треба знайти елемент. Попередня ініціалізація допомагає зекономити час на чотирьох кроках.

4.4 Створення багатопотоковості та запуск автоматизованих тестів на виділеному сервері

Для забезпечення багатопотокового (паралельного) запуску тестів необхідно створити .xml файл для опису потоків, у які буде запущено тести. В залежності від потужності машини, на якій система тестування буде працювати, можна створювати різну кількість потоків. Саме тому, запуск автоматизованих тестів на виділеному сервері прискорює тестування.

В .xml файлі за допомогою попереднього розділення сценаріїв тестування на різні класи описуються потоки виконання тестування. На рисунку 4.4 видно, що кожен клас буде запущено окремим потоком, тобто кожен сценарій тестування буде відбуватися незалежно від іншого в один і той самий час.

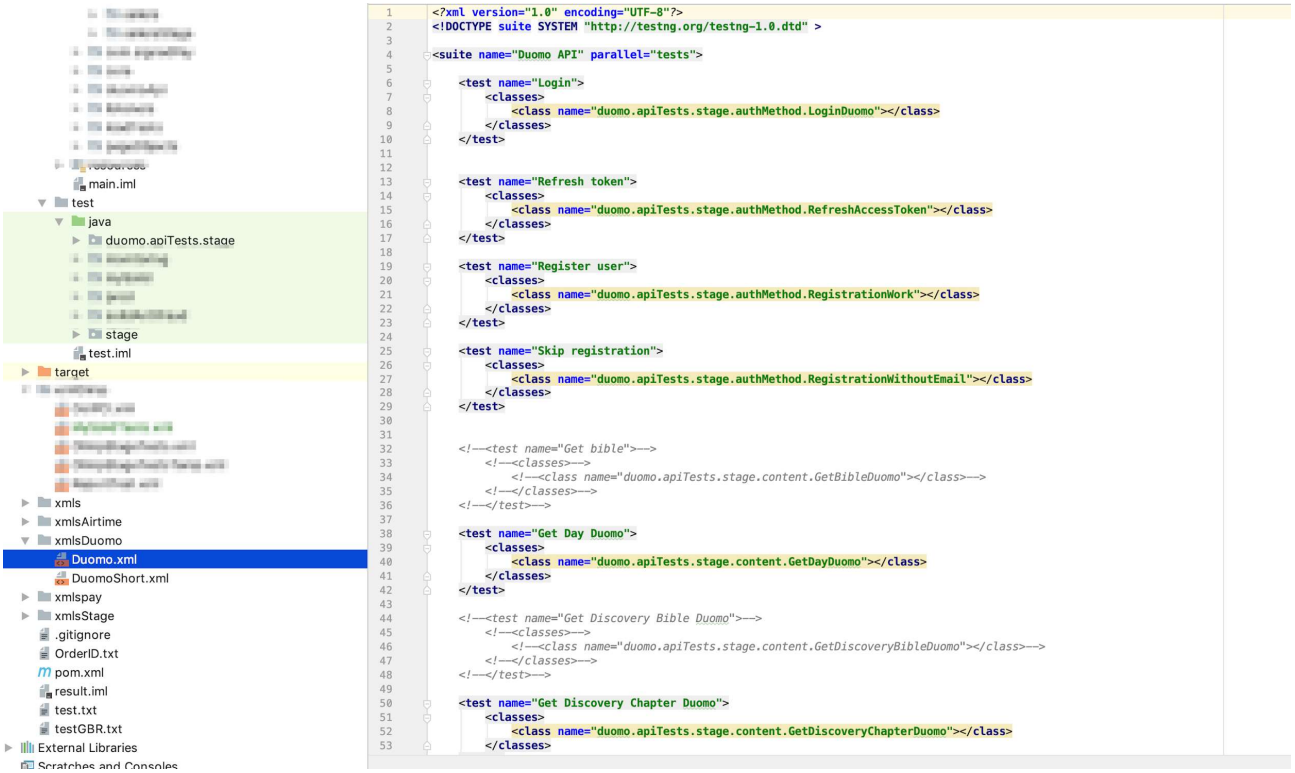


Рисунок 4.4 - Створення .xml файлу для опису потоків автоматизованого тестування

За результатами запуску системи автоматизованого тестування на прикладі мобільного додатку “Duomo”, що була розроблена на основі створеної методики прискорення тестування можна зазначити, що час виконання автоматизованих тестів було скорочено більше, ніж в чотири рази (рис. 4.5) з сорока дев’яти хвилин до одинадцяти. При цьому кількість та якість виконання тестів залишились незмінними.

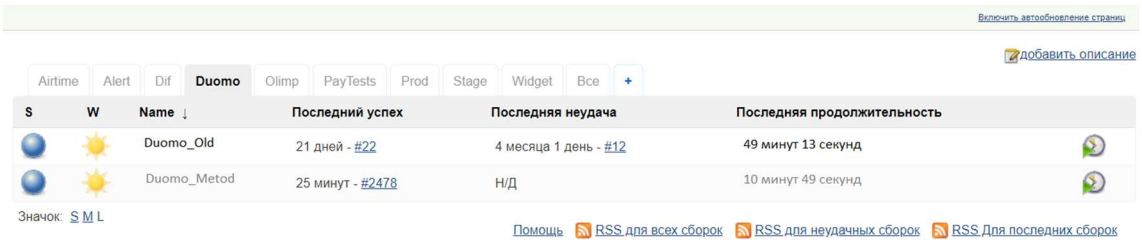


Рисунок 4.5 – Порівняльний час виконання автоматизованих тестів

Висновки до 4 розділу

За результатами проведеного практичного тестування методики прискорення автоматизованого тестування мобільних додатків, можна стверджувати, що методика працює та дійсно дає змогу прискорити тестування програмного продукту. Отже, задача розробки методики а також задача створення автоматизованих тестів на основі методики, виконані в повному обсязі.

ВИСНОВКИ

У ході роботи розроблено методику, яка значно прискорює автоматизоване тестування мобільних додатків. У роботі була вирішена задача розробки підходів, яка у свою чергу, дозволяє прискорити написання автоматизованих тестів та їх виконання, що, в свою чергу, допомагає створити певний рівень абстракції між інструментами розробника та графічним інтерфейсом мобільного додатку. Програма має представляти графічні елементи як окремі об’єкти та надавати тестувальнику можливість швидко та якісно проводити тести і мати можливість легко їх змінювати.

Вивчено проблеми тестування мобільних додатків. Однією з головних проблем автоматизованого тестування є його трудомісткість: попри те, що воно дозволяє усунути частину рутинних операцій і прискорити виконання тестів, великі ресурси можуть витрачатися на оновлення самих тестів. З іншого боку, при зміні інтерфейсу програми необхідно заново переписати всі тести, які пов'язані з оновленими вікнами, що при великій кількості тестів може відняти значні ресурси. Другим недоліком є повільне виконання автоматизованих тестів при великому обсязі роботи та великій кількості тестів.

З точки зору особливостей поставленої проблеми для визначення можливостей вирішення задачі прискорення тестування мобільних додатків було проведено аналіз існуючих методів. Також було проаналізовано недоліки та переваги існуючих засобів автоматизованого тестування.

На основі проведеного аналізу було обрано методи та інструментальні засоби для створення методики прискорення автоматизованого тестування мобільних додатків.

В роботі представлено детальний опис методики та інструментальних засобів, необхідних для її використання. Методика може бути використана при тестуванні будь-яких мобільних додатків.

Було виконано практичну перевірку результативності методики. За результатами перевірки, можна стверджувати, що методика працює та дійсно дає змогу прискорити тестування програмного продукту.

Задачі, поставлені на початку даної роботи, виконані в повному обсязі. На даний момент методика прискорення тестування та система автоматизованого тестування мобільного додатку «Duomo» на її основі впроваджена в роботу ТОВ «Айті Ленд», на чиє замовлення була виконана робота.

ЛІТЕРАТУРА

1. IEEE Guide to Software Engineering Body of Knowledge, SWEBOK, 2004

2. Голощаов А. Л. Google Android програмування для мобільних пристроїв / А. Л. Голощаов. – Санкт-Петербург, 2011.

3. Хашими С. Розробка додатків для Android / С. Хашими, М. Коматинени. – Санкт-Петербург, 2011.

4. Android 2. Програмування додатків для планшетних комп'ютерів та смартфонів (Рето Майер, Эксмо, 2011)

5. GitHub – Iamtodor/Diabetes: App for people, who have diabetes diseases [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/Iamtodor/Diabetes

6. Android Developers [Електронний ресурс] – Режим доступу до ресурсу: https :// developer. android. com / index.html

7. Android Developers Blog [Електронний ресурс] – Режим доступу до ресурсу: http :// android – developers. blogspot. com/

8. Stack Overflow [Електронний ресурс] – Режим доступу до ресурсу: http :// stackoverflow. com/

9. Search — Medium [Електронний ресурс] – Режим доступу до ресурсу: https :// medium. com / search ? q =android

10. XML – Wikipedia, the free encyclopedia [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/XML

11. Методология функционального моделирования SADT [Електронний ресурс] – Режим доступу до ресурсу: http://www.infosystem.ru/designing/methodology/sadt/theory_sadt.html.

12. Хомоненко А.Д., Цыганков В.М., Мальцев М.Г. Базы данных: Учебник для высших учебных заведений/Под ред. проф. А.Д. Хомоненко. – СПб.: КОРОНА принт, 2002. – 672с.

13. Братищенко В.В. Проектирование информационных систем. — Иркутск: Изд-во БГУЭП, 2004. — 84 с.

14. Магазин Google Play [Електронний ресурс] – Режим доступу до ресурсу: https://uk.wikipedia.org/wiki/Google_Play.

15. Інтернет-портал присвячений створенню додатків для системи Android. -2012. Режим доступу: http://habrahabr.ru/post/109944/. - Дата доступу: 28.04.2013.

16. Голощапов, А.Л. Google Android програмування для мобільних пристроїв / А.Л. Голощапов - М.: BHV Санкт-Петербург, 2011. - 1549 с.

17. Хашимі, С. Розробка додатків для Android / С. Хашимі. - М.: Біном, 2011. - 2125 с.

18. Дерсі, Л. Android за 24 години. Програмування додатків під операційну систему Google / Л. Дерсі. - Рід Груп, 2011. - 1499с.

19. Еккель, Б. Філософія Java, 4-е видання / Б. Еккель. - М.: Біном, 2009. - 1768 с.

20. Рік Роджерс, Джон Ломбардо, «Android Розробка додатків», ЕКОМ Паблишерз, ISBN 978-5-9790-0113-5, 978-0-596-52147-9; 2010 р

21. Голощапов А.Л. Google Android: Програмування для мобільних пристроїв. - СПб .: БХВ-Петербург, 2011 року.

22. Програмування для Android. Самовчитель / Колісниченко Д. - СПб .: Санкт-Петербург, 2011. - 736 с.

23. Android 2. Програмування додатків для планшетних комп'ютерів і смартфонів / Рето Маєр. - СПб .: Санкт-Петербург, 2011. - 672 с.

24. Статті про програмування для Android [Електронний ресурс] // URL: http://flashbot.ru/android-dev

25. Офіційна довідка по середовищі програмування [Електронний ресурс] // URL: http://www.jetbrains.com

26. Форум про програмування для Android [Електронний ресурс] // URL: http://www.cyberforum.ru/android-dev/

27. Форум про програмування для мобільних пристроїв [Електронний ресурс] // URL: http://www.4pda.ru

28. Програмування під Android / Блейк Мік. - СПб .: Санкт-Петербург, 2012. - 496 с.

29. Смартфони Android без напрягу. Керівництво користувача / Андрій Жвалевский. - СПб .: Санкт-Петербург, 2012. - 224 с.

30. Програмування для Android. Самовчитель / Денис Колісніченко. - СПб .: Санкт-Петербург, 2011. - 272 с.

31. Соммервілла, І. Інженерія програмного забезпечення / пер. з англ. А.А. Мінько, А.А. Момотюк, Г.І. Сингаївська. - М .: ВИЛЬЯМС, 2002. - 624 с., Іл.

32. Тестування програм / В.В. Липа. - М .: Радио и связь, 1986. - 437 с.

33. Брауде, Е. Технологія розробки програмного забезпечення / пер. з англ .. - СПб .: ПИТЕР, 2004. - 655 с., іл.

34. Орлов, С. Технології розробки програмного забезпечення / С.А. Орлов. - СПб .: ПИТЕР, 2002. - 464 с., Іл.

35. Липа, В. Надійність програмних засобів. - М .: Сінтег, 1998. - 358 с.

36. Вігерс, К. Розробка вимог до програмного забезпечення / пер. з англ .. - М .: «Російська Редакція», 2004. - 576 с., іл.

ДОДАТОК А

.xml файл паралелізації тестування

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="Duomo API" parallel="tests">

  <test name="Login">
    <classes>
      <class name="duomo.apiTests.stage.authMethod.LoginDuomo"></class>
    </classes>
  </test>


  <test name="Refresh token">
    <classes>
      <class name="duomo.apiTests.stage.authMethod.RefreshAccessToken"></class>
    </classes>
  </test>

  <test name="Register user">
    <classes>
      <class name="duomo.apiTests.stage.authMethod.RegistrationWork"></class>
    </classes>
  </test>

  <test name="Skip registration">
    <classes>
      <class name="duomo.apiTests.stage.authMethod.RegistrationWithoutEmail"></class>
    </classes>
  </test>

  <test name="Get Day Duomo">
    <classes>
      <class name="duomo.apiTests.stage.content.GetDayDuomo"></class>
    </classes>
  </test>

  <test name="Get Discovery Chapter Duomo">
    <classes>
      <class name="duomo.apiTests.stage.content.GetDiscoveryChapterDuomo"></class>
    </classes>
  </test>

  <test name="Get Discovery Duomo">
    <classes>
      <class name="duomo.apiTests.stage.content.GetDiscoveryDuomo"></class>
    </classes>
  </test>

  <test name="Get Discovery Letter Duomo">
    <classes>
      <class name="duomo.apiTests.stage.content.GetDiscoveryLetterDuomo"></class>
    </classes>
  </test>

  <test name="Get Discovery One Time Action">
    <classes>
      <class name="duomo.apiTests.stage.content.GetDiscoveryOneTimeAction"></class>
    </classes>
```

```
    </test>

    <test name="Get Faq Duomo">
        <classes>
            <class name="duomo.apiTests.stage.content.GetFaqDuomo"></class>
        </classes>
    </test>

    <test name="Get Feedback Duomo">
        <classes>
            <class name="duomo.apiTests.stage.content.GetFeedbackDuomo"></class>
        </classes>
    </test>

    <test name="Get Habit Duomo">
        <classes>
            <class name="duomo.apiTests.stage.content.GetHabitDuomo"></class>
        </classes>
    </test>

    <test name="Get Journey Chapter Duomo">
        <classes>
            <class name="duomo.apiTests.stage.content.GetJourneyChapterDuomo"></class>
        </classes>
    </test>

    <test name="Get Journey Duomo">
        <classes>
            <class name="duomo.apiTests.stage.content.GetJourneyDuomo"></class>
        </classes>
    </test>

    <test name="Get Journey Full Duomo">
        <classes>
            <class name="duomo.apiTests.stage.content.GetJourneyFullDuomo"></class>
        </classes>
    </test>

    <test name="Get Letter Duomo">
        <classes>
            <class name="duomo.apiTests.stage.content.GetLetterDuomo"></class>
        </classes>
    </test>

    <test name="Get One Time Action Duomo">
        <classes>
            <class name="duomo.apiTests.stage.content.GetOneTimeActionDuomo"></class>
        </classes>
    </test>

    <test name="Get Profile Duomo">
        <classes>
            <class name="duomo.apiTests.stage.content.GetProfileDuomo"></class>
        </classes>
    </test>

    <test name="Get Questions Duomo">
        <classes>
            <class name="duomo.apiTests.stage.content.GetQuestionsDuomo"></class>
        </classes>
    </test>
</suite>
```

ДОДАТОК Б

Лістинг системи автоматизованого тестування мобільного додатку «Duomo» на основі розробленої методики

```
package uiDuomo.iosUITests;

import duomoIOS.iosScenariosSteps.JourneyIosSteps;
import org.junit.Assert;
import org.junit.Test;

public class JourneyFirstIosTestScenario {

  static JourneyIosSteps journeyIosScenario;


  @Test
  public void testJ() {
    journeyIosScenario.initJourneyIosPageObject();
    journeyIosScenario.clickDiscoverySectionFromJourney();
    journeyIosScenario.waitDiscoveryDisplayed();
    Assert.assertTrue(journeyIosScenario.isDiscoveryDisplayed());
  }

  @Test
  public void testK() {
    journeyIosScenario.initDiscoveryIosPage();
    journeyIosScenario.clickJourneySectionFromDiscovery();
    Assert.assertTrue(journeyIosScenario.isJourneyDisplayed());
  }

  @Test
  public void testL(){
    journeyIosScenario.initJourneyIosPageObject();
    journeyIosScenario.clickFirstJourneyChapter();
    journeyIosScenario.waitChapterIntro();
    Assert.assertTrue(journeyIosScenario.isChapterIntroDisplayed());
  }

  @Test
  public void testM(){
    journeyIosScenario.initChapterIntroPage();
    journeyIosScenario.clickIntroBackBtn();
    Assert.assertTrue(journeyIosScenario.isJourneyDisplayed());
  }

  @Test
  public void testN(){
    journeyIosScenario.initJourneyIosPageObject();
    journeyIosScenario.clickFirstJourneyChapter();
    journeyIosScenario.waitChapterIntro();
    journeyIosScenario.initChapterIntroPage();
    journeyIosScenario.clickIntroContinueBtn();
    Assert.assertTrue(journeyIosScenario.isChapterDaysDisplayed());
  }

  @Test
  public void testO() {
    journeyIosScenario.initChapterIosPageObject();
  }
}
```

```
package duomoAndroid.screenObjects;

import io.appium.java_client.AppiumDriver;
import org.openqa.selenium.WebElement;

public class LandingPageObject {

   public WebElement signUpBtn;
   public WebElement logInLink;
   public WebElement continueWithFb;
   public WebElement textView;

   public LandingPageObject(AppiumDriver driver) {
      signUpBtn = driver.findElementById("btn_continue");
      logInLink = driver.findElementById("ll_signin");
      continueWithFb = driver.findElementById("ll_facebook");
      textView = driver.findElementById("text_view");
   }


}
package duomoAndroid.screenObjects;

import io.appium.java_client.AppiumDriver;
import org.openqa.selenium.WebElement;

public class PreQuestionsPageObject {

   public WebElement introText;
   public WebElement introName;
   public WebElement beginTheJourneyBtn;


   public PreQuestionsPageObject(AppiumDriver driver) {
      introText = driver.findElementById("text_intro");
     //  introName = driver.findElementById("text_name");
      beginTheJourneyBtn = driver.findElementById("btn_start_journey");

   }
}
package duomoAndroid.screenObjects;

import io.appium.java_client.AppiumDriver;
import org.openqa.selenium.WebElement;

public class SignUpPageObject {

   public WebElement signUpBtn;
   public WebElement nameLine;
   public WebElement emailLine;
   public WebElement passLine;
   public WebElement backBtn;

   public SignUpPageObject (AppiumDriver  driver) {
      signUpBtn = driver.findElementById("btn_signup");
      nameLine = driver.findElementById("input_name");
      emailLine = driver.findElementById("input_email");
      passLine = driver.findElementById("input_password");
      backBtn = driver.findElementById("world.duomo.duomo:id/ivBack");
   }



}
```

```
package duomoAndroid.scenariosAndroidSteps;

import duomoAndroid.core.DriverStart;
import duomoAndroid.screenObjects.*;
import org.openqa.selenium.By;
import org.openqa.selenium.support.ui.ExpectedConditions;


public class SignUpStepsAndroid extends DriverStart {

  SignUpPageObject signUpPageObject;
  LandingPageObject landingPageObject;
  PreQuestionsPageObject preQuestionsPageObject;
  QuestionsPageObject questionsPageObject;
  FacebookPageObject facebookPageObject;
  PaymentShopPageObject paymentShopPageObject;

  //Landing Page

  public void initLandingPageAndroid() {
    landingPageObject = new LandingPageObject(driver);
  }

  public void waitLandingLoad() {
    driverWait.until(ExpectedConditions.visibilityOfElementLocated(By.id("btn_continue")));
  }

  public boolean isLandingPageDisplayed() {
    return driver.findElementById("world.duomo.duomo:id/ll_signin")
        != null;
  }

  public void clickContinueWithFbLanding() {
    landingPageObject.continueWithFb.click();
  }


  public void clickSignUpBtnLanding() {
    landingPageObject.signUpBtn.click();
  }

  public void waitSwipeLeft() {
    driverWait.until(ExpectedConditions.textToBePresentInElement(By.id("text_view"),
        "Incorporate Christian values into your behaviors and traits"));
  }

  public void waitSwipeRight() {
    driverWait.until(ExpectedConditions.textToBePresentInElement(By.id("text_view"),
        "Make God a part of your everyday life in just 10 minutes a day"));
  }

  public void clickTextViewLanding() {
    landingPageObject.textView.click();
  }

  // Facebook WebView Object

  public void waitFbOpen() {
    driverWait.until(ExpectedConditions.visibilityOfElementLocated(By.id("m_login_email")));
  }

  public boolean isfbOpen() {
    return driver.findElementByAndroidUIAutomator("m_login_email")
        != null;
  }
```

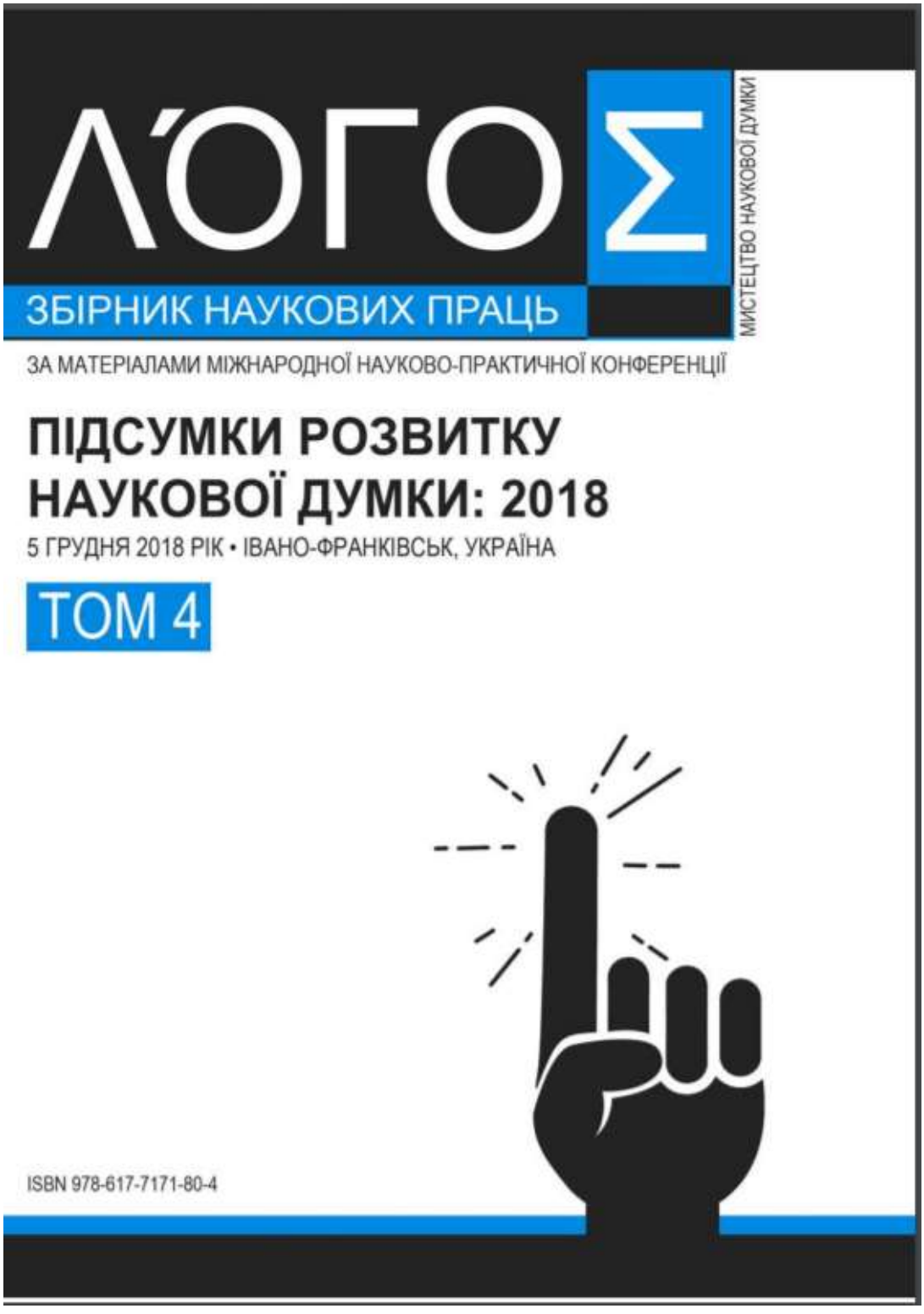
```
public void initFacebookPage() { facebookPageObject = new FacebookPageObject(driver); }

public void clickCloseFbBtn() {
    facebookPageObject.closeFbBtn.click();
}

//Sign Up Page Object

public boolean isSignUpPageDisplayed() {
    return driver
                .findElementByAndroidUIAutomator("new UiSelector().textContains(\"Create Account\")")
                != null;
}

public void initSignUpPage() {
    signUpPageObject = new SignUpPageObject(driver);
}

public void clickBackBtnSignUp() {
    signUpPageObject.backBtn.click();
}

public void inputName(String name) {
    signUpPageObject.nameLine.click();
    signUpPageObject.nameLine.sendKeys(name);
}

public void inputEmail(String emailAuto) {
    signUpPageObject.emailLine.click();
    signUpPageObject.emailLine.sendKeys(emailAuto);
}

public void inputPass(String passAuto) {
    signUpPageObject.passLine.click();
    signUpPageObject.passLine.sendKeys(passAuto);
}

public void confirmSignUpBtn() {
    signUpPageObject.signUpBtn.click();
}

//Pre Question Page Object

public void waitPreQuestionPage() {
    driverWait.until(ExpectedConditions.visibilityOfElementLocated(By.id("text_intro")));
}

public boolean isPreQuestionPageDisplayed() {
    return driver
          .findElementById("text_intro")
          != null;
}

public void initPreQuestionPage() {
    preQuestionsPageObject = new PreQuestionsPageObject(driver);
}

public String getIntroText() {
    return preQuestionsPageObject.introText.getText();
}

public String getIntroName() {
    return preQuestionsPageObject.introName.getText();
}
```

```
    public void clickBeginJourneyBtn() {
        preQuestionsPageObject.beginTheJourneyBtn.click();
    }

    //Questions Page Object

    public void initQuestionsPage() {
        questionsPageObject = new QuestionsPageObject(driver);
    }

    public String getInformText() { return questionsPageObject.informText.getText(); }

    public String getFirstAnswerText() {
        return questionsPageObject.answerOne.getText();
    }

    public void clickFirstAnswer() {
        questionsPageObject.answerOne.click();
    }

    public String getSecondAnswer() {
        return questionsPageObject.answerTwo.getText();
    }

    public void clickSecondAnswer() {
        questionsPageObject.answerTwo.click();
    }

    public String getThirdAnswer() {
        return questionsPageObject.answerThree.getText();
    }

    public void clickThirdAnswer() {
        questionsPageObject.answerThree.click();
    }

    //Payment Shop Page Object

    public void waitShopPage() {
        driverWait.until(ExpectedConditions.visibilityOfElementLocated(By.id("sale2")));
    }

    public boolean isPaymentPageDisplayed() {
        return
                driver
                        .findElementById("sale2")
                        != null;
    }

    public void initShopPage() {
        paymentShopPageObject = new PaymentShopPageObject(driver);
    }

    public void clickCloseShopBtn() {
        paymentShopPageObject.closeShopBtn.click();
    }

    //Journey


}
```

ДОДАТОК В

Участь у конференції

Мобільний додаток для підвищення розумової діяльності шляхом фізичних навантажень / О.Д. Зубарчук, В.В. Козяр // Підсумки розвитку наукової думки: 2018, Збірник тез, м. Івано-Франківськ, Україна, 5 грудня.



МОБІЛЬНИЙ ДОДАТОК ДЛЯ ПІДВИЩЕННЯ РОЗУМОВОІ ДІЯЛЬНОСТІ ШЛЯХОМ ФІЗИЧНИХ НАВАНТАЖЕНЬ

Зубарчук Олександра Дмитрівна
Науковий керівник: канд. т. наук Козяр Василь Васильович
Національний технічний університет України
«Київський політехнічний інститут імені І. Сікорського»
Україна

Однією з особливостей сучасного суспільства є тривале реформування його суспільно-виробничої сфери, яке глибоко і не завжди позитивно змінює весь спосіб життя, праці і побуту людини.

Сучасна навчальна діяльність та діяльність працівників різних сфер характеризується зростанням обсягу інформації, з одного боку, і, при цьому, низьким рівнем рухової активності, одноманітністю робочої пози та робочих процесів. У зв'язку з цим багатьма вченими і практиками приймаються спроби вирішення питання про те, що необхідно збалансувати підвищені навантаження в розумовій діяльності і оптимальну рухову активність.

Фізіологічні та психологічні основи розумової активності досліджували автори посібника В. Г. Ткачук і В. Є. Хапко у своїх роботах [1, 2]. Згідно із цими дослідженнями необхідність чергування розумової праці та відпочинку є однією з фізіологічних особливостей людини. Режими праці та відпочинку базуються на науковому підґрунті з урахуванням фізіологічних закономірностей пристосування організму людини до умов життя.

Відомо, що постійне нервово-психічне перенапруження і хронічне розумова перевтома без фізичного відпочинку викликають важкі функціональні розлади в організмі та зниження рівня розумової активності людини. Саме тому, регулярні фізичні навантаження мають бути невід'ємною складовою життя людини в сучасному світі.

На основі вищевикладеного була визначена мета роботи: інтенсифікація розвитку розумових здібностей людей різного віку за рахунок підвищення рухової активності. Для досягнення даної мети було розроблено мобільний додаток для операційної системи Android з персональним мобільним тренером для підвищення фізичної активності та тестами на розумову активність для відслідковування впливу фізичних навантажень на підвищення рівню розумової діяльності.

Тести для визначення рівню розумової активності були розділені на 4 категорії:

1. Сприйняття (тест «переплутані лінії»);
2. Увага (тест коректурної проби);
3. Пам'ять (запам'ятовування матеріалу);
4. Мислення (розв'язування судоку).

Додаток розроблявся під операційну систему Android з наступних причин:

1. Кількість користувачів операційною системою Андроїд на сьогодні є найбільшою.
2. Зручна та інтуїтивно зрозуміла середа розробки Android Studio;
3. Java - поширена мова програмування (розробка під ОС iOS вимагає знань специфічних мов програмування);

4. Можливість швидкого доступу до готового додатку за рахунок передачі .apk файлів та прямого встановлення додатку на пристрій;

5. Відсутність обмежень в засобах розробки (для розробки під ОС iOS обов'язково необхідно мати ПК з ОС MacOS);

6. Безкоштовність (для розробки під ОС iOS необхідно платити щорічний внесок у розмірі $99);

Взаємодія користувача з мобільним додатком для підвищення розумової активності шляхом фізичних навантажень передбачає наступні кроки (рис. 1):

1. Запуск додатку на мобільному девайсі з ОС Андроїд версії 5.0 та вище;

2. Реєстрація в системі. Введення персональних даних таких як: вік користувача, його зріст та вага (для визначення індексу маси тіла) та рівень фізичної підготовки.

3. Початок роботи з додатком. Проходження первинного тесту для визначення рівню розумової активності користувача на основі;

4. Робота з персональною програмою тренувань та тестами на визначення рівню розумової активності;

5. Перегляд прогресу рівню розумової активності завдяки підвищення фізичних навантажень.

Рис. 1. **Схема взаємодії користувача з додатком для підвищення розумової активності шляхом фізичних навантажень**

На діаграмі послідовностей (рис. 2) відображено основні послідовності дій користувача з інтерфейсом, а також показано відправку запитів і отримання відповідей з backend частини додатку для режимів виконання фізичних вправ та перегляду результатів.

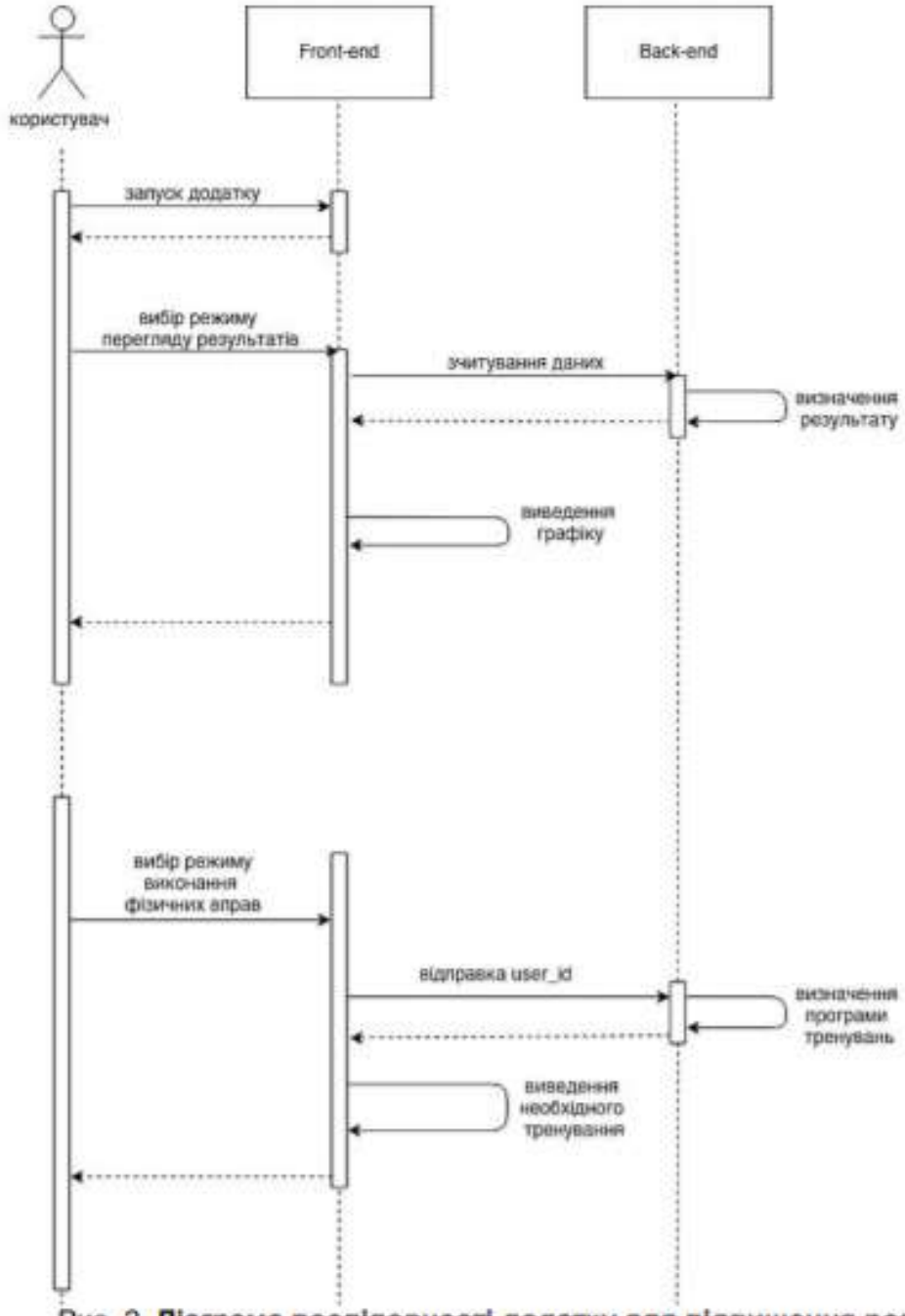


Рис. 2. **Діаграма послідовності додатку для підвищення розумової активності людини шляхом фізичних навантажень**

Після реалізації додатку було проведено експеримент серед 60 осіб, що були розділені на 5 рівномірних груп:

- жінки віком від 18 до 22 років
- чоловіки віком від 18 до 22 років
- жінки віком від 23 до 29 років
- чоловіки віком від 23 до 29 років
- чоловіки віком від 30 до 45 років

Усереднений за п'ятьма групами графік зростання рівня розумової активності зображено на рисунку 3. Лініями синього кольору на графіку позначено рівень уваги в учасників експерименту, червоного - сприйняття, жовтого - пам'ять та зеленого - рівень мислення.

Показники різних категорій незалежні та мають різний рівень оцінювання при проходженні тестів, тому не можна стверджувати, що рівень мислення вищий від рівню уваги чи сприйняття.

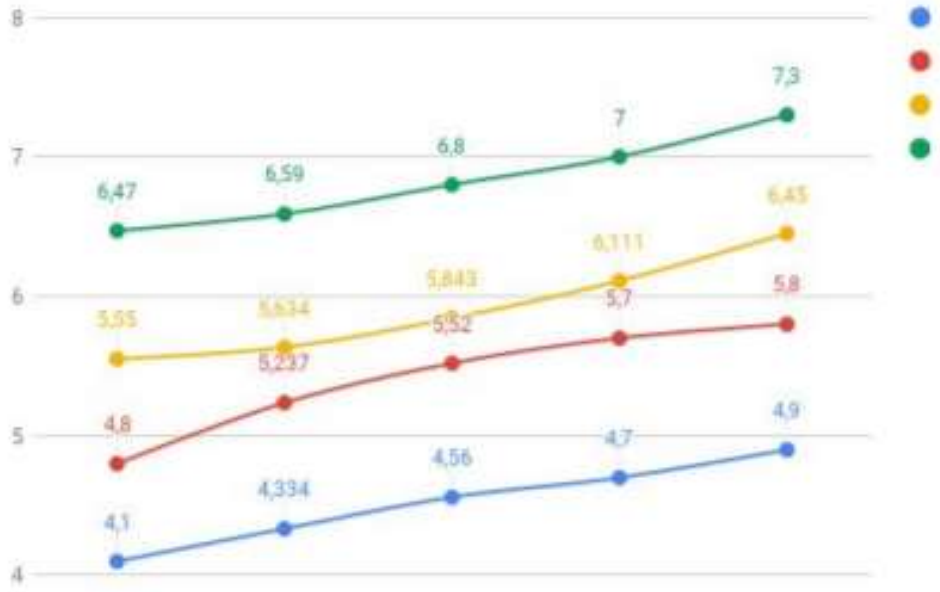


Рис. 3. **Графік зростання рівня розумової активності серед учасників експерименту**

На основі результатів експерименту можна зробити наступні висновки: після 2 тижнів регулярних занять фізичними вправами можна побачити позитивну зміну показників рівня розумової активності осіб усіх груп. Також варто зазначити, що всі учасники експерименту помітили підвищення настрою та загального стану здоров'я.

Список використаних джерел:

1. Ткачук В. Г., Хапко В. Е. Психофізіологія праці. К. : МАУП, 2000.
2. Ткачук В. Г., Хапко В.Е. Анатомия и эволюция нервной системы. Краткий конспект лекций. МАУП, 2010.